DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CALIFORNIA 93943-5002

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

TEST AND EVALUATION OF
THE TRANSPUTER
IN A MULTI-TRANSPUTER SYSTEM

by

Jose Vanni Filho

June 1987

Thesis Advisor U.R. Kodres

Approved for public release; distribution is unlimited.

T233113

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Aproved for public release; distribution is unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b OFFICE SYMBOL (If applicable): 52	7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO; PROJECT NO; TASK NO; WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
Test and Evaluation of The Transputer in a Multi-Transputer System

12 PERSONAL AUTHOR(S)
Jose Vanni Filho

13a TYPE OF REPORT: Master's Thesis	13b TIME COVERED: FROM ____ TO ____	14 DATE OF REPORT (Year Month Day): 1987, June	15 PAGE COUNT: 201

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GROUP	Parallelism, Concurrency, Distributed Systems, Performance Evaluation, Transputer, Occam

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

The purpose of this thesis is to start the evaluation of the Transputer, a 32 bit microprocessor on a chip, to verify its potentials and limitations for real time applications, in distributed systems.

The evaluation concentrates on the four physical communication links, and its advertised capability to operate in parallel with the main processor (CPU), each one of them at rate of 10 mbit/sec in each direction. It also presents to the reader an introduction to the machine itself, to the Occam Programming Language, a description of the environment at the Naval Postgraduate School(NPS), and suggests to the novice a learning sequence.

The evaluation programs and other example programs presented in this thesis were implemented using the Occam Programming Language (Proto-Occam) in either the Occam Programming System (OPS) or the Transputer Development System (TDS), both resident on the VAX 11/780 computer under the VMS Operating System (VAX/VMS).

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL: Uno R. Kodres	22b TELEPHONE (Include Area Code): (408) 646 2197; 22c OFFICE SYMBOL: 52Kr

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited.

Test and Evaluation of the Transputer in a Multi-Transputer System

by

Jose Vanni Filho
Lieutenant Commander, Brazilian Navy
B.S., Brazilian Naval Academy, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1987

ABSTRACT

The purpose of this thesis is to start the evaluation of the **Transputer,** a 32 bit microprocessor on a chip, to verify its potentials and limitations for real time applications, in distributed systems.

The evaluation concentrates on the four physical communication links, and its advertised capability to operate in parallel with the main processor (CPU), each one of them at rate of 10 mbit/sec in each direction. It also presents to the reader an introduction to the machine itself, to the Occam Programming Language, a description of the environment at the Naval Postgraduate School(NPS), and suggests to the novice a learning sequence.

The evaluation programs and other example programs presented in this thesis were implemented using the Occam Programming Language (Proto-Occam) in either the Occam Programming System (OPS) or the Transputer Development System (TDS), both resident on the VAX 11/780 computer under the VMS Operating System (VAX/VMS).

+4275

THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis are listed below the firm holding the trademark:

Digital Equipment Corporation, Maynard, Massachusetts

- VAX 11/780Minicomputer.
- VMS Operating System
- VT-220 Terminal
- VT-100 Terminal

Hewlett Packard Corporation

- Hewlett Packard
- HP

INMOS Group of Companies, Bristol, UK

- Transputer
- Occam
- INMOS
- IMS

Intel Corporation, Santa Clara, California

- 86/12A Single Board Computer (SBC)
- MULTIBUS Architecture
- 8086, 80286, 80386 microprocessors

International Business Machines Corporation, Boca Raton, Florida

- IBM
- IBM PC

Tektronix Inc., Beverton, Oregon

- Tektronix

United States Government

Ada Programming Language

Xerox Corporation, Stanford, Connecticut

Ethernet

Zenith Data Systems Corporation, St. Joseph, Michigan

Z-248 Micro-computer

TABLE OF CONTENTS

I. INTRODUCTION 14
A. BACKGROUND 14
1. Intended Audience 15
B. WHY THE TRANSPUTER 15
C. THE OCCAM PROGRAMMING LANGUAGE 18
1. Primitives 19
2. Constructs 20
3. Good Features of Proto-Occam 22
4. Proto-Occam Limitations 23
D. THE ENVIRONMENT AT THE NPS 23
1. Software Facilities 23
2. Hardware Facilities 24
E. STRUCTURE OF THE THESIS 27

II. COMMUNICATION AND PERFORMANCE ISSUES 29
A. COMMUNICATION ISSUES 29
1. Definitions: 29
2. Data Transmission Basics 29
B. THE TRANSPUTER LINKS 30
C. EXPECTED RESULTS 31
1. One Channel Transmitting 32
2. Both Channels Transmitting/Receiving 32
D. RESEARCH QUESTIONS 32
E. PERFORMANCE MEASUREMENT ISSUES 33
1. Hardware Methods 34
2. Software Methods 35

III. THE EVALUATION STARTS 37
A. INTRODUCTION 37

1. The Available Constructs 37
2. Considerations About Memory Management 38
B. A CLOSE LOOK ON THE BIT RATE 38
1. First Software Results 39
2. Using the Oscilloscope 40
3. Comparison Between the Constructs 46
C. OBSERVING PARALLEL ACTIVITY ON THE LINKS 49
1. Using Software 49
2. Using the Oscilloscope 53
3. Using the Logic Analyzer 55
4. Comparison Between the Four Constructs 58
D. MESSAGE SIZE AND CHANNEL PARALLELISM INFLUENCE 59
1. How to Read the Tables 59
2. BYTE SLICE Procedure 61
3. WORD SLICE Procedure 63
4. Input and Output Primitives 63

IV. THE MUTUAL EFFECTS BETWEEN PROCESSOR AND THE FOUR LINKS 67
A. EFFECT OF CONCURRENT PROCESSES OVER COMMUNICATIONS 67
1. Initial Considerations 67
2. Process Priority Considerations 68
3. BYTE SLICE Procedure 69
4. WORD SLICE Procedure 75
5. Input and Output Primitives 78
B. THE EFFECT OF THE COMMUNICATIONS OVER CONCURRENT PROCESSES 80
1. Initial Considerations 80
2. Results Obtained 82
C. DOES THE TRANSPUTER ACHIEVE LINEAR PERFORMANCE IMPROVEMENTS? 85

V. CONCLUSION 89

APPENDIX A: LEARNING SEQUENCE 92
a. How to Log in 92
b. Learning Sequence 92

APPENDIX B: OPS TUTORIAL 95

APPENDIX C: TDS TUTORIAL 100

APPENDIX D: HINTS ABOUT OCCAM PROGRAMMING 104
a. Program Structure 104
b. Problems and Suggestions 105
c. Comments About the Link Evaluation Program 107

APPENDIX E: THE LINK EVALUATION PROGRAM 109

APPENDIX F: PROGRAM TEST LINEARITY 186

APPENDIX G: TRANSPUTER PRODUCTS* 194
a. Transputers 194
b. Evaluation Boards 194
c. Digital Signal Processing 194

LIST OF REFERENCES 195

BIBLIOGRAPHY 197

INITIAL DISTRIBUTION LIST 198

LIST OF TABLES

1. TRANSPUTER T-414 TECHNICAL DATA AND CHARACTERISTICS 16
2. PROCESSOR CYCLE TIME/CLOCK EXAMPLES 17
3. CHARACTERISTICS OF BOARDS B001, B003 AND B004 25
4. EXPECTED MAXIMUM TRANSFER RATES ON THE TRANSPUTER LINKS 32
5. THE DIFFERENT TICK VALUES 36
6. MAXIMUM TRANSFER RATES OBTAINED (KBITS/SEC) 47
7. LINK MAP FOR FIGURE 3.21 58
8. EFFECT OF PARALLELISM ON TRANSFER RATES FOR 10000 BYTES BLOCK SIZE ** 58
9. TRANSPUTER LINK TRANSFER RATE BYTE SLICE (1) - NO CONCURRENT PROCESS - 10 MBITS/SEC 60
10. TRANSPUTER LINK TRANSFER RATE BYTE SLICE (2) - NO CONCURRENT PROCESS - 10 MBITS/SEC 61
11. TRANSPUTER LINK TRANSFER RATE - WORD SLICE - NO CONCURRENT PROCESS - 10 MBITS/SEC 63
12. TRANSPUTER LINK TRANSFER RATE - INPUT/OUTPUT (BYTES 1) - NO CONCURRENT PROCESS - (10 MBITS/SEC) 64
13. TRANSPUTER LINK TRANSFER RATE - INPUT/OUTPUT (BYTES 2) - NO CONCURRENT PROCESS - (10 MBITS/SEC) 65
14. TRANSPUTER LINK TRANSFER RATE - INPUT/OUTPUT (WORDS 1) - NO CONCURRENT PROCESS - (10 MBITS/SEC) 66
15. TRANSPUTER LINK TRANSFER RATE - INPUT/OUTPUT (WORDS 2) - NO CONCURRENT PROCESS - (10 MBITS/SEC) 66
16. TRANSPUTER LINK TRANSFER RATE - BYTE SLICE - PROCEDURE CPUBUSYSUM CONCURRENT AT THE B003 - 10MBITS/SEC 70

17. TRANSPUTER LINK TRANSFER RATE - BYTE SLICE - PROCEDURE CPUBUSYPROD CONCURRENT AT THE B003 - 10MBITS/SEC 70
18. TRANSPUTER LINK TRANSFER RATE - BYTE SLICE - PROCEDURE CPUBUSYSUM CONCURRENT AT ALL CPUS - 10MBITS/SEC 72
19. TRANSPUTER LINK TRANSFER RATE - BYTE SLICE - PROCEDURE CPUBUSYSUM CONCURRENT AT THE B003 (HIGH) - 10 MBITS/SEC 74
20. TRANSPUTER LINK TRANSFER RATE - BYTE SLICE PROCEDURE CPUBUSYSUM CONCURRENT AT ALL CPUS (HIGH) - 10 MBITS/SEC 75
21. NUMBER OF OPERATIONS EXECUTED CONCURRENTLY IN EACH CPU*- BYTE SLICE USED 75
22. TRANSPUTER LINK TRANSFER RATE* - INPUT/OUTPUT (BYTES) PROC CPUBUSYSUM CONCURRENT - 10 MBITS/SEC 78
23. TRANSPUTER LINK TRANSFER RATE* - INPUT/OUTPUT (WORDS) PROC CPUBUSY.SUM CONCURRENT - 10 MBITS/SEC 78
26. TIMMING OF PROCEDURE COUNTER 84
25. COMPARING COUNTER EXECUTION TIME IN 4 AND 16 TRANSPUTERS NETWORK 88

LIST OF FIGURES

1.1 Block Diagram of Transputer Architecture 18
1.2 System Using a Transputer as Memory 19
1.3 Example of a PAR Construct 21
1.4 Example of an ALT Construct 21
1.5 Replicated PAR 22
1.6 Replicated ALT 23
1.7 The Four Transputers in the B003 Board - Fixed Links 24
1.8 System Interconnections VAX-Transputers-Terminal 26
2.1 The Data and Acknowledge Frames 31
3.1 The BYTE SLICE OUTPUT Procedure Call 37
3.2 Basic Code for Transmitter and Receiver 39
3.3 Configuration for Initial Tests 40
3.4 Configuration for Measuring Links at 20 mbits/sec Bit Rate 40
3.5 Frame Transmitted for Oscilloscope Observations 41
3.6 Example Code for Oscilloscope Observations 41
3.7 Picture of One Frame at 10mbits/sec Rate 42
3.8 Three Data Frames at 10mbits/sec Rate 43
3.9 Five Frames Observed at 10 mbits/sec Rate 44
3.10 One Frame and the ACK at 20 mbits/sec Rate 45
3.11 Four Frames and ACK at 20 mbits/sec Rate 46
3.12 TRUES Transmitted Using the Input/Output Primitives 48
3.13 Maxint Transmitted Using the Input/Output Primitives 49
3.14 Configuration to Observe the Four Links Operating in Parallel 50
3.15 Code Used to Time Transmission Through the Four Links in Parallel 51
3.16 Code for the Receivers 51
3.17 Configuration Code for the Link Evaluation Program 52
3.18 Two Channels of Different Links Transmitting at the Same Time 53
3.19 Two Channels of the Same Link Operating at the Same Time 54

3.20 Output from the Logic Analyzer of 4 Channels in Parallel 56
3.21 8 Channels Monitored with the Logic Analizer 57
3.22 Transputer Link Transfer Rate Byte Slice - No Process in Parallel - 10 mbits/sec 62
4.1 CPU modes Available in the Link Evaluation Program 67
4.2 How the Concurrent Processes Were Called 68
4.3 Transputer Link Transfer Rate - BYTE SLICE Procedure Cpubusysum Concurrent at the B003 - 10 mbits/sec 71
4.4 Transputer Link Transfer Rate - BYTE SLICE Procedure Cpubusysum Concurrent at All CPUs - 10 mbits/sec 73
4.5 Transputer Link Transfer Rate - BYTE SLICE Procedure Cpubusysum Concurrent at the B003(high) - 10 mbits/sec 76
4.6 Transputer Link Transfer Rate - BYTE SLICE Procedure Cpubusysum Concurrent at All CPUs(high) - 10 mbits/sec 77
4.7 Procedure Counter 80
4.8 Configuration for Program Test Linearity (17) 81
4.9 Procedure Route 83
4.10 Procedure Route5 86
4.11 Structure of Program Test Linearity (5) 87
A.1 Keypad for Using the Fold Editor 94
B.1 OPS Utilities 97
B.2 First Program in OPS 99
C.1 The Utilities for the TDS System 103
D.1 OPS Program Structure 104
D.2 TDS Program Structure Example 105
D.3 SKIP Usage 106

DEDICATION

This thesis is dedicated to:

my wife Edwiges and
our children Viviane, Guilherme and Denise.

I. INTRODUCTION

A. BACKGROUND

The NPS AEGIS project has in its primary goals the research and development of alternative architectures for the AEGIS Combat Weapon System (CWS), focusing on low cost, reliable and fault tolerant architectures. As the cost of micro-processors has been decreasing incredibly and the capabilities are always increasing, it turns out to be very attractive to think of using these cheap and powerful tools to accomplish the functions of any system.

One branch of this research is based upon the Intel 86/12A Single Board Computers that are working under the MCORTEX operating system [Ref. 1], fully developed at the AEGIS lab. It exploits the 10mbits/second capacity of the Intel MULTIBUS and uses the concept of shared memory to allow multiprocessors arranged in clusters of up to eight single board computers, to increase the throughput of the system. Each cluster has its own shared memory whose access is controlled by means of eventcounts. The clusters intercommunicate through an Ethernet link [Ref. 2].

One alternative concept for distributed systems is the use of message passing [Ref. 2,3]. The Transputer concept exploits this idea and produces a very interesting and flexible way of designing multiprocessor systems. This second branch of research is now in its third released work[1] and is increasing in importance and extent.

This thesis was developed in parallel and concurrently with the one from Cordeiro, M. M. [Ref. 6]. Since these theses were in fact the first to really program this new machine, at the NPS, a series of obstacles were encountered and overcome one by one, up to the point we were able to divide the work, and on our own, search for the answers we were individually seeking. This is the reason why we tried to point out many of the pitfalls that one may encounter in future research in this area using the system available at the NPS.

---

[1]See B. Evin Implementation of A Serial Delay Insertion Type Loop Communication for a Real Time Multi-Transputer System [Ref. 4] and Selcuk, Z., Implementation of a Serial Communication Process for a Fault Tolerant, Real Time, Multi-Transputer Operating System [Ref. 5].

1. Intended Audience

This will be a good first reading for the person beginning to work with transputers or Occam. Appendix A presents a Learning Sequence; Appendix B presents an OPS Tutorial; Appendix C presents a TDS Tutorial and Appendix D provides some hints on how to program in Occam. It also will be a good reference for transputer users and real-time system designers and implementors in the sense of what they can expect in terms of performance from the INMOS links. They will be able use the Tables, Graphics and the Evaluation Programs to check and confirm their expectations in issues concerning what should be the right construct or the right message size to use, in order to achieve the desired throughput or communication rate.

B. WHY THE TRANSPUTER

The TRANSPUTER is a member of a family of micro-processors, that have on one chip, the processor, its own local memory and links for point to point connections to other transputers.

Each transputer product contains special circuitry and interfaces adapting it to each particular use. For example a peripheral control transputer, such as a graphics or disk controller, has interfaces tailored to the requirements of a specific device [Ref. 7].

The transputers were designed in parallel with the Occam programming language and were first released in 1985. Now, two years later, there is a growing variety of transputers available on the market with different capabilities and for different applications. Some of these are listed in Appendix G.

The T-414 is a 32 bit micro-processor with 2 kbytes of on chip RAM, four standard INMOS serial links, external memory interface and peripheral interfacing on a single 1.5 micron CMOS chip. As an example, its characteristics and technical data are summarized in Table 1, and its internal architecture is depicted in Figure 1.1[2].

For the sake of comparison, Table 2 lists the processor cycle time or internal clock of other commercially available computers and also some processors used in military applications for real-time.

The internal architecture of the transputer follows Von Newman principles and permits the processor itself to run at the same time as the 4 links operate. This way a high level of parallelism is achieved already on chip level.

---

[2]Reproduced by permission of INMOS Corporation.

TABLE 1

TRANSPUTER T-414 TECHNICAL DATA AND CHARACTERISTICS

	processor cycle time	internal clock	instruction throughput
(T 414-20)	50 nsec	20 mhz	10 MIPS
(T 414-15)	67 nsec	15 mhz	7.5 MIPS
(T 414-12)	80 nsec	12.5 mhz	6.0 MIPS

external clock cycle	5 mhz
time slice	1 msec (approximately)
internal bus speed	80 mega bytes/second
internal (on chip) memory	2 kilo bytes
internal memory cycle	50 nsec (for 50 nsec cpu)
external memory interface	25 mega bytes/second bus
external memory cycle	150 nsec
address capability	4 giga bytes(32 bit address)
links (serial)	4 (full duplex, DMA)
link bit rate	10 mbits/sec (20 mbits/sec)
link net bit rate (Obs. 1)	3.8 mbits/sec (6.1 mbits/s)
power dissipation	less than 500 milliwatts
physical dimensions	45 mm square chip (84 pins)

Obs. 1: These values refer to the immediately above mentioned bit rates, respectively, and are fully explained on chapter III.

When reading transputer related material, one may find references to T-424. This was a prototype that is not on the market anymore.

The systems architecture is simplified by using the transputer links for point to point communications which allows the available transputers to be configured in any desired topology matching the programmer needs [Ref. 4,8]. Point to point communication links have many advantages over multiprocessor buses [Ref. 7]:

- There is no contention for the communication mechanism, regardless of the number of transputers in the system (that does not happen in shared memory systems) [Ref. 9].

TABLE 2

PROCESSOR CYCLE TIME/CLOCK EXAMPLES

SBC 86/12A		(1 to 8 mhz)
80286 -		(6 to 12 mhz)
80386 -		(16 mhz)
Transputer T 414-20...	50 nsec	(20mhz)
AN/UYK 7 -	750 nsec	
AN/UYK 43 -	150 nsec	
VAX 11/780 -	200 nsec	
IBM 3033 -	57 nsec	
IBM 3081 (k) -	26 nsec	

- There is no capacitive load penalty as transputers are added to a system. (specially if they will work independently)

However as the number of transputers increase in the system, a message routing system is needed in order to permit indirectly interconnected transputers to communicate to each other. This will create some overhead for the system and Cordeiro [Ref. 6], addresses this point presenting a design and implementation for such a system.

It is up to the programmer to decide which process should be placed in which processor. For efficiency purposes, it is recommended to place frequently communicating processes in adjacent transputers (directly connected by a link).

It is still possible however, to adapt previously designed systems, to this new architecture and develop a systems architecture where a central data base would be managed by a central transputer, which would address a large memory that could be read or written by the processors connected to the four links, as depicted in Figure 1.2 . But this would involve further study and it is not in the scope of this thesis.

Another point worthy to mention is that although this work was developed using OCCAM, there is already available a C[3] compiler, and coming soon a Pascal and a Fortran compiler. The Ada compiler for this machine is under development and according to INMOS representatives, it will be released before the end of 1987.

This way the final goal of the AEGIS project, that is to research alternative ways of implementing the AEGIS system will have the DoD language available on the present machine.

[3]The C programming language compiler generates code for the transputer

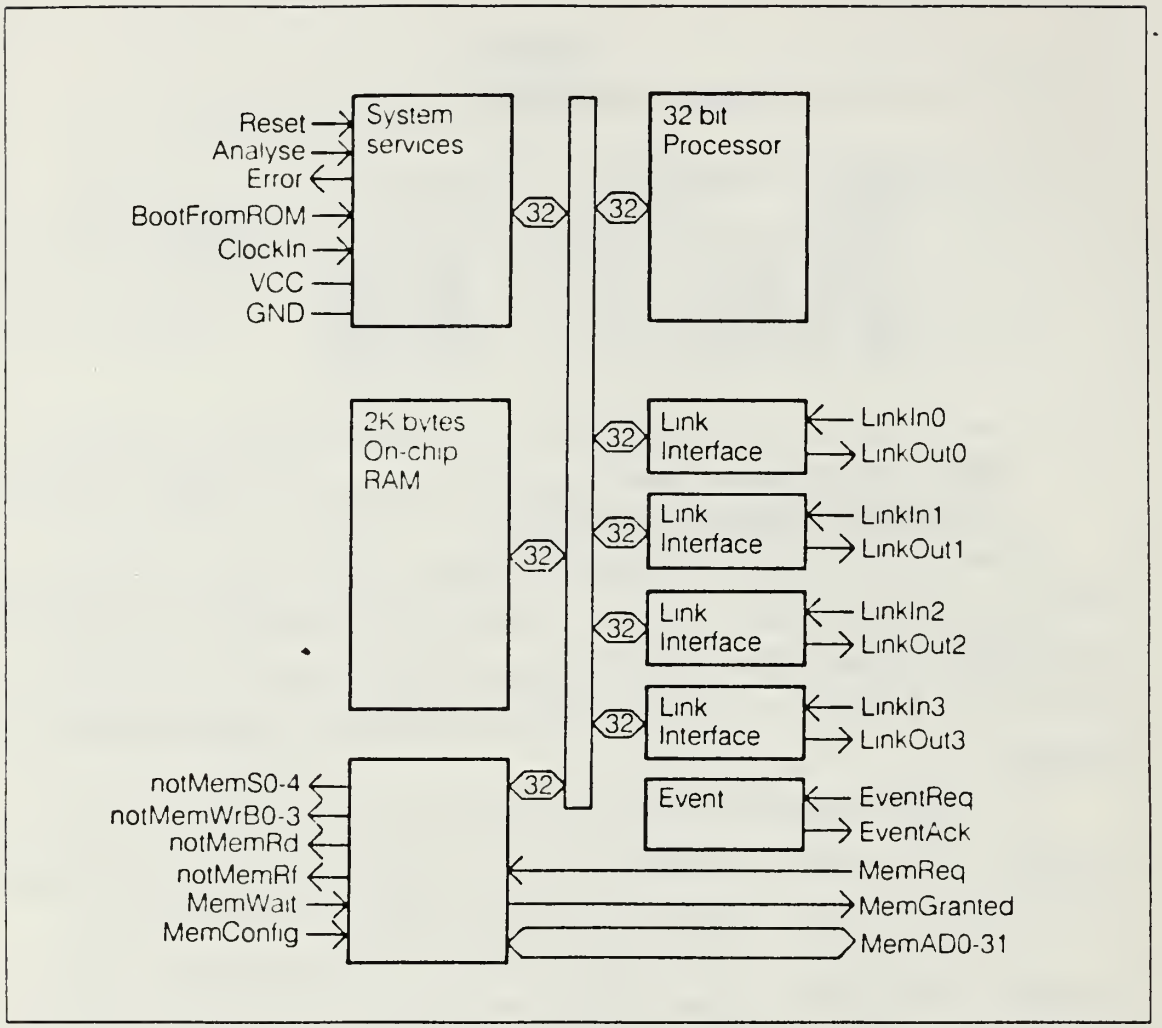


Figure 1.1 Block Diagram of Transputer Architecture.

It is also pertinent to mention at this point that in the last Occam User Group meeting, that took place in Santa Clara, CA, in March 10th, 1987 there were representatives of IBM, Tektronix and other major corporations showing to the participants, work in development and developed by them, using the Transputer.

C. THE OCCAM PROGRAMMING LANGUAGE

Occam is a programming language that since its first release in 1983 is known as very suitable for description of multiple processor systems [Ref. 10], due to the simplicity with concurrency and parallelism can be addressed [Ref. 11].

In fact, since then, the language has been modified and enhanced in its capabilities, and one of the latest versions, known as **Occam 2** is described in the book

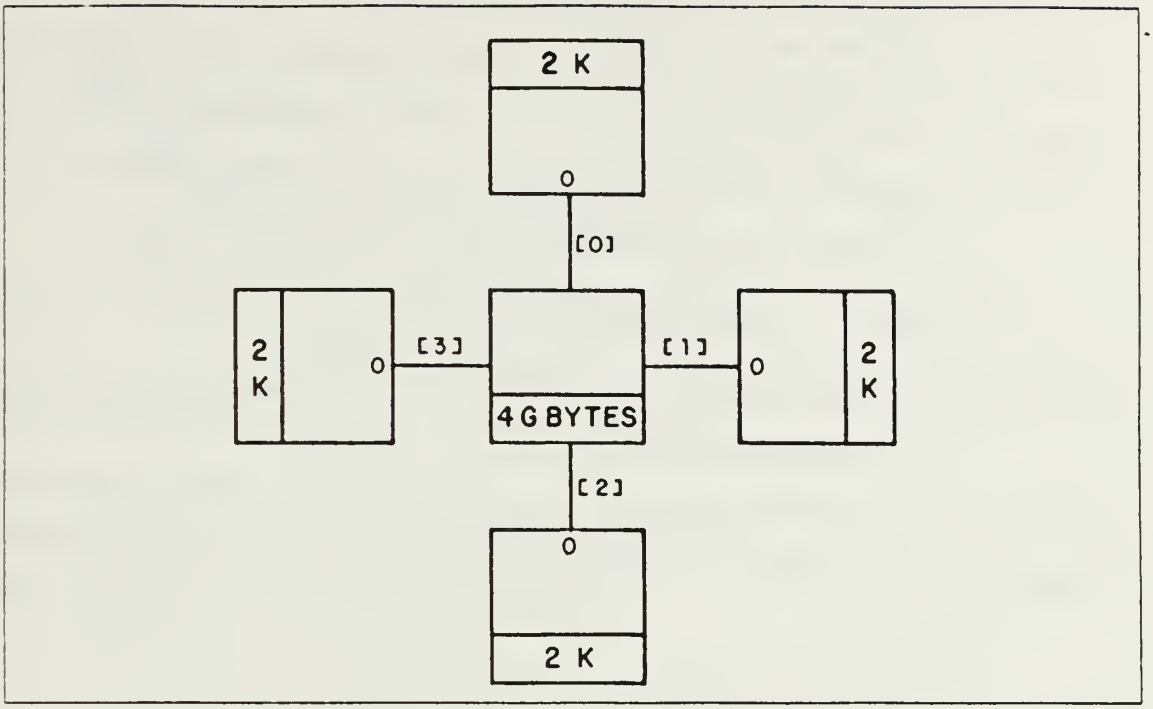


Figure 1.2 System Using a Transputer as Memory.

by Pountain [Ref. 12]. However, this thesis was developed using one of the primitive versions of the language called **Proto-Occam** [4] that is best described in the Occam Programming Manual [Ref. 13: section 3], with slight modifications introduced by the OPS TDS compilers implementations described in detail in the Occam Implementation section in the OPS Manual [Ref. 13: section 4].

The goal of this section is to address briefly the primitive processes and constructs used in Occam (Proto-Occam), calling attention to the limitations and capabilities this version of the language has. Appendix D presents some hints for programming the transputer T-414 using Proto-Occam.

1. **Primitives**

a. The Channel

The channel (CHAN) is an identifier used for performing communications between concurrent processes (if in the same processor) or processes executed in parallel (if in different processors). We can think of the channels as a pipe that connects horizontally two processes that are being executed concurrently or in parallel.

[4]Proto-Occam is so called in the Occam Programming Manual, but sometimes it is also referenced as being **Occam 1** .

If the processes are in the same processor (same transputer), this is done through a specified memory location determined at compile time, as if it were a global variable; but if the communicating processes are in different transputers, the channel uses the physical links connecting the transputers. Any type of variable may flow through the channel, but the programmer must ensure that the type being transmitted is the same that is being expected at the receiver, or the compiler will flag an error.

This is the basic for the primitives **input** and **output:**

- chanin ? char - This can be read as the variable "char" will receive a value that is coming from elsewhere through the channel chanin.
- chanout ! 5 - This can be read as the constant "5" is being output to another process through the channel chanout.

This implies that somewhere in our transputer network there will be a process that is transmitting some value through the channel "chanin" and another (or may be the same) process is receiving into some variable the value "5" through a channel called chanout.

2. **Constructs**

Occam has six basic constructs:

a the sequential (SEQ) construct
b the parallel (PAR) construct
c the alternative (ALT) construct
d the conditional (IF) construct
e the repetitive (WHILE) construct
f the replicators (FOR) construct.

The sequential, conditional and repetitive constructs have the same usage as in many other structured languages.[5] It is interesting to note the necessity of having a SEQ construct, because normally in such languages this is the only way to execute a program.

a. The PAR Construct

A parallel construct causes its component processes to be executed in parallel, if the component processes reside on different transputers, or concurrently in a time shared fashion, if they reside on the same processor [Ref. 13: section 3, item 3.4.2].

Note from Figure 1.3 that:

- Process one and process two are different processes.
- Occam is fixed format and indentations are always 2 spaces for nesting.

[5]Like Pascal, Ada or C programming languages.

```
CHAN comms, c1, c2 :   --- channel declarations
PAR
  WHILE TRUE      --- process one
    VAR x :
    SEQ
      c1 ? x
      comms ! x    --- end process one
  WHILE TRUE       --- process two
    VAR y :
    SEQ
      comms ? y
      c2 ! y       --- end process two
```

Figure 1.3 Example of a PAR Construct.

- There are no begins or ends to delimit processes.
- We can declare variables anywhere in the code as long as it is before the beginning of the process that will refer to it.
- Three dashes (---) are the indication for comments following them.

(1) *The PRI PAR Construct.* The priority parallel construct, a variation of the PAR construct, permits at most two processes under it. The first one will be given priority 0 (high), and the second one will be given priority 1 (low). This maps exactly to the two priority levels that the chip supports. As the Reference Manual [Ref. 7: p. 3], says, the priority process is expected to be executing for a short period of time because when it begins, it can not be preempted.

b. The ALT Construct

An alternative construct is used to accept the first message available from a number of **input channels** [Ref. 13: section3,item 3.4.3]. See Figure 1.4 .

```
CHAN c1, c2 :
WHILE TRUE
  VAR x :
  ALT
    C1 ? x
      c3 ! x
    c2 ? x
      c3 ! x
```

Figure 1.4 Example of an ALT Construct.

Note from Figure 1.4 that:

- We could have any number of channels under the ALT and all of them outputing to c3. This is a construct that provides **mutual exclusion** [6] in two lines of code.
- All variable declarations are separated by commas and terminated by a colon.

There is also a variation of the ALT construct named PRI ALT, that enables the first option of the ALT be executed in precedence to the others.

c. Replicators

A replicator may be used with a construct SEQ, PAR, ALT or IF to replicate the process a number of times [Ref. 13: section 3, item 3.4.6]:

- SEQ - When used with SEQ it provides a conventional loop.
- PAR - When used with a PAR it makes an array of concurrent processes See Figure 1.5
- ALT - When used with ALT it enables to receive one unique input at a time from an array of channels. See Figure 1.6 .

```
CHAN c[n+1] :
PAR i = [0 FOR n]
  WHILE TRUE
    VAR x :
    SEQ
      c[i] ? x
      c[i + 1] ! x
```

Figure 1.5 Replicated PAR.

3. **Good Features of Proto-Occam**

Proto-Occam has some nice features like:

- the facility in handling time for performance evaluation (TIME ? var)
- the use of time delay (TIME ? AFTER sometime) for real-time applications
- the SKIP that has numerous applications and help to handle exceptions
- we can access the byte in memory
- there is no need to declare count variables used in replicators

[6]Mutual exclusion is one of the critical issues in Operating System design [Ref. 3] and it is neatly handled by the ALT construct.

```
CHAN c[n], d :
WHILE TRUE
  VAR y :
  ALT i = [ 1 FOR n ]
    c[i] ? y
      d ! x
```

Figure 1.6 Replicated ALT.

- we can have procedures with formal parameters being arrays of variable size; this way the actual parameters may be of different sizes in different procedure calls.

4. **Proto-Occam Limitations**

Many of the limitations of Proto-Occam have been fixed by Occam 2, but they are still note worthy:

- there are only one dimensional arrays
- there are no types; the programmer has to establish a convention to use its variable names and make sure to address them coherently.
- no floating point is available
- no recursion is permitted
- no pointers are available

D. THE ENVIRONMENT AT THE NPS

1. Software Facilities

The Naval Postgraduate School has several Software tools available in its computer labs:

- Occam Programming System (OPS), available in the VAX/VMS. It permits editing, compiling, linking and running on the VAX, concurrent programs written in Occam, simulating a network of transputers. It will be briefly described in Appendix B, but the reader may refer to [Ref. 13].
- Transputer Development System (TDS D600), available for the VAX/VMS, it edits, compiles and down loads the code into the transputer network. It will be briefly described in Appendix C, but additional information may be obtained in [Ref. 14].
- Transputer Development System (TDS D701), available for PC-AT type micro-computers. It edits, compiles, links, and down loads to the transputer network the code to be executed (that was generated on the PC). It is single user and requires installation of the B004[7] board in the PC. It uses the Occam 2

[7]Described in Table 3.

programming language. This system arrived at the lab at a point in time that this thesis was already partially written and so it will not be addressed. For more information refer to [Ref. 15].

2. Hardware Facilities

a. Transputer Boards

The transputer lab has a Transputer Evaluation Module with seventeen (17) transputers in the following configuration:

- one board with one transputer (T414-12) called B001 [Ref. 16], that is the interface with the VAX/VMS.
- four boards with four transputers (T414-15) each called B003 [Ref. 17], that can be used either with the VAX or with the PC.

It also has one board with one transputer (T414-15) called B004 [Ref. 18], that is the interface with the PC, and is located in one of the slots of the Zenith Z-248. This makes a total of 18 transputers to work with.

Table 3 lists its characteristics.

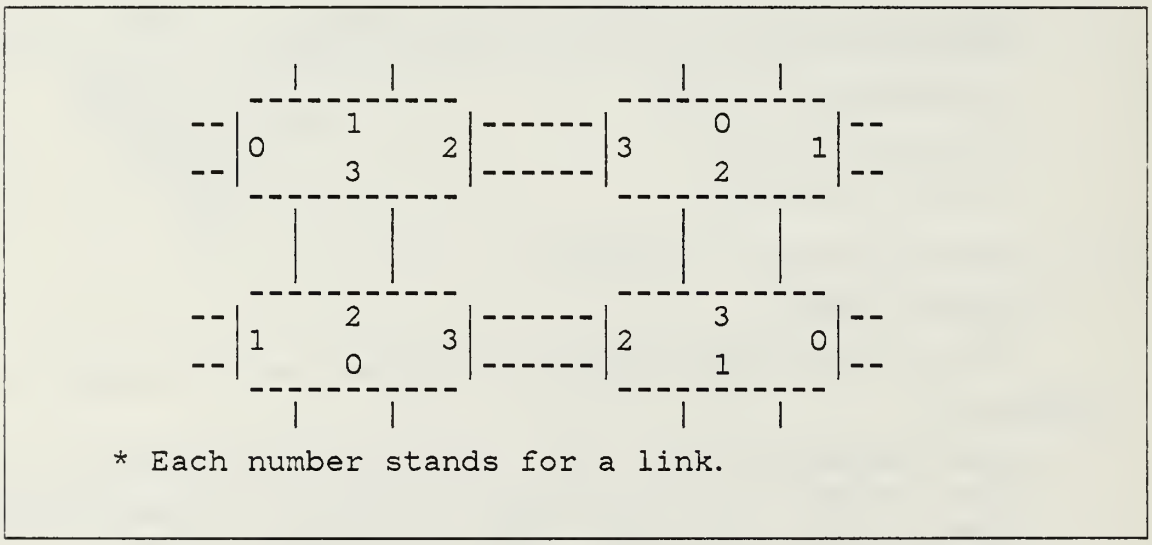


Figure 1.7 The Four Transputers in the B003 Board - Fixed Links.

These transputers can be interconnected and configured in any way designed by the programmer using the INMOS links as long as the hard wired board connections between transputers (that already exits and are fixed in all B003 boards in the LAB) are respected [Ref. 17]. See Figure 1.7 .

TABLE 3

CHARACTERISTICS OF BOARDS B001, B003 AND B004

a. B001 Board
- One IMS T 414 - 12 mhz transputer
- 10 mbits/sec INMOS link transmission speed
- 64 kbytes of static RAM (32 x IMS 1400-45)
- 128 kbytes EPROM (4 x 27256) containing :
  .bootstrap loader,
  .memory test,
  .terminal to host transparent mode software
- 2 RS/232 serial input/output connectors for :
  .VAX connection
  .Terminal connection
- 64 way DIN connector for external link connections

b. B003 Board
- 4 IMS T 414-15 mhz transputers
- 10 or 20 mbits/sec INMOS link transmission speed
- 256 kbytes dynamic RAM per transputer
- 96 way DIN connector for external link connections

c. B004 IBM-PC Add-in-Board
- one T 414-15 mhz transputer
- 10 mbits/sec INMOS link transmission speed
- 2 mbytes dynamic RAM with parity
- 62 pin I/O channel connector

The B001 board is the interface between the VAX and the transputer network. The interconnection is done through standard RS 232.

The user can develop OCCAM programs on the VAX, debug and test using the OPS, and when ready, down load them to be run on the transputers. See Appendix D.

b. Host Computers and Terminals

(1) *VAX.* To use any of the systems (TDS or OPS) on the VAX, the user must log in from any VT 100 or VT 220 terminal (this last one has to be in VT 100 mode, and VT 100 id). Appendix A presents a detailed sequence for this.

To be able to down load the executable code, the terminal must be also connected to it. There are two ways of doing it [Ref. 16,] and Figure 1.8 shows how this is done at the NPS lab.

The following advantages should be pointed out:

- The VAX provides us with the VMS Operating System and all the facilities a mini-computer can support, mainly a weekly system backup that we do not have to worry about.

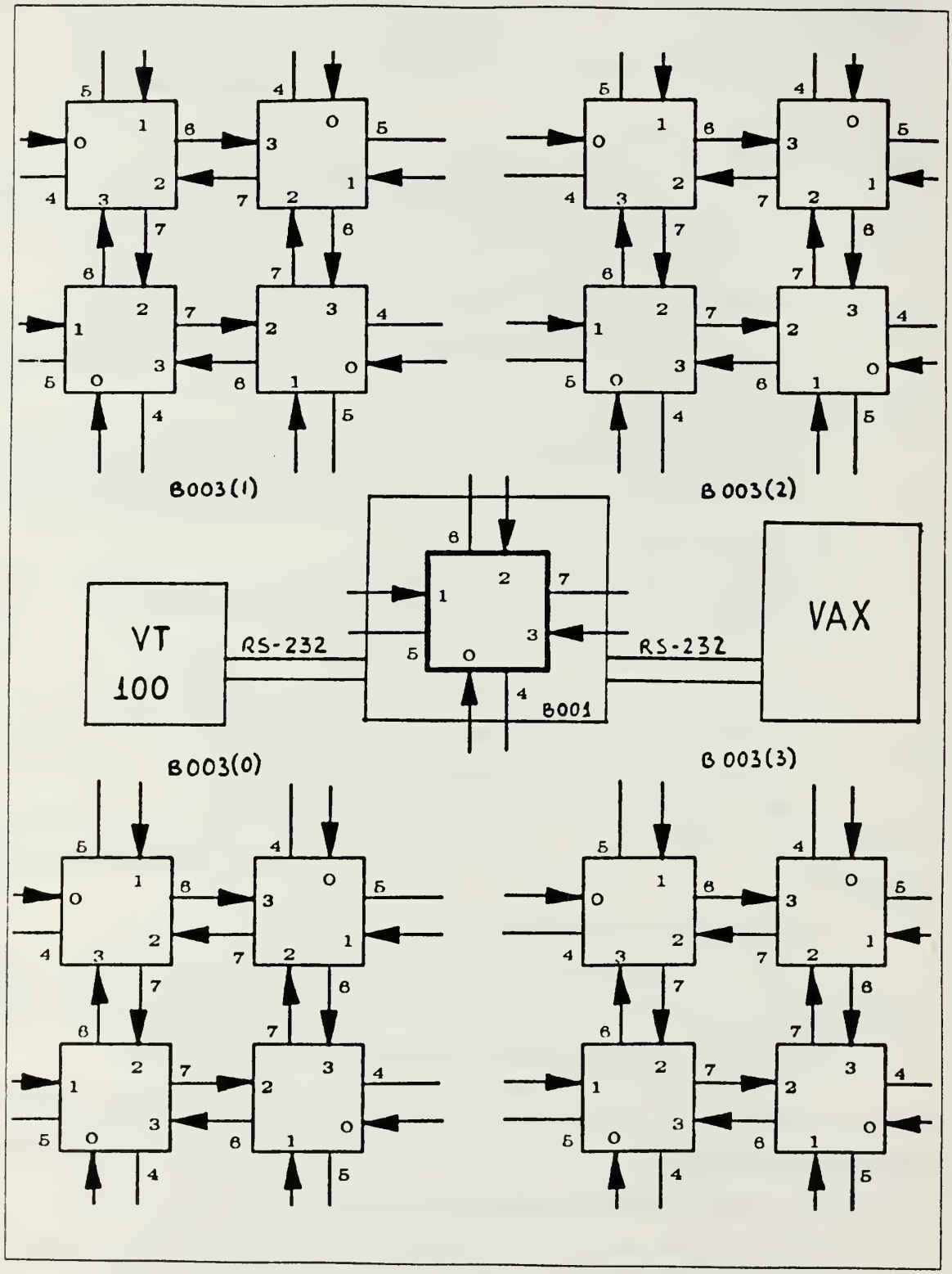


Figure 1.8 System Interconnections VAX-Transputers-Terminal.

- If a modem is available, much of the work can be done from home.
- Multi-user utilization as far as using the OPS and editing and compiling in the TDS (Very handy for class projects).
- Occam 2 will be available soon in the VAX at the NPS, as an upgrade of the OPS and TDS systems.

The only disadvantage is that when the VAX is down for backup, upgrades or repairs, there is nothing the user can do about it.

(2) *Zenith Z-248.* The TDS system for the PC is completely independent from the VAX. It has a new version of OCCAM more powerful and flexible. It is installed in a Zenith Z-248 micro-computer (PC-AT compatible), with 2.5 mbytes of RAM and 8 MHz clock.

There are two advantages in having a PC

first the user has the whole micro for him and no problems, except a TDS system failure, would delay any project. An assumption is made that to replace a PC is an easy task. Secondly, the Occam version running on the PC is temporarily[8] newer than the one on the VAX, and new horizons are opened for research.

As mentioned before, this thesis was developed on the TDS and OPS installed on the VAX and it will not have any other information on the PC based system.

c. Printing Facilities

There are two ways to print OCCAM programs developed on the VAX:

- Using the VAX / VMS online printer (only files with extensions ".lst" and ".lis" are printable).
- Using the printer at the lab and the print screen facility provided by the VT 220 terminals. Anything that is on the screen can be printed this way, and this turned out to be one of the best debugging and analyzing tools for the research.

E. STRUCTURE OF THE THESIS

This thesis is presented in 5 Chapters and 7 Appendixes.

Chapter I was the introduction to Occam, the transputer and the NPS environment. Chapter II describes the terminology, the INMOS Links, the methods used for performance evaluation, and state the expected results and research questions.

Chapter III and Chapter VI address each one of the research questions, describing the experiments done and presenting the results obtained and conclusions reached thereto. Chapter V summarizes the conclusions and suggests future research.

---

[8]The Occam 2 version for the VAX/VMS will be available at any moment.

As mentioned already, Appendix A presents a Learning Sequence for how to work with the transputers and Occam, having the VAX/VMS System as a host. Appendix B and Appendix C, are tutorials about the software tools available presently for the VAX , the OPS and the TDS systems.

Appendix D presents some hints in how to program in Occam, and call attention for some mistakes that most likely one will make when using this new language on a new system, with a different and powerful fold editor.

Appendix E lists the Link Evaluation Program used, and Appendix F lists the Test Linearity Program, both with all procedures and library routines that were used. When reading the listing files take into account that :

- Occam is a fixed indentation language with two spaces between each nested level.
- Two dashes (--) marks the begining of new folder with the title aside.
- Three dashes (---) means that comments follow on that line only.

II. COMMUNICATION AND PERFORMANCE ISSUES

A. COMMUNICATION ISSUES

The purpose of this section is to set the stage and define a series of communication terms that will be used in the following discussion about the transputer physical links performance.

1. **Definitions:**

- frame - it is a packet of bits containing 8 bits plus the frame protocol bits (e.g. start bit, stop bit, and parity bit).
- bit rate - it is the number of bits that can be transmitted in a unit of time (e.g. kbits/sec or mbits/sec).
- baud rate - is the number of signal elements transmitted per second. If there are only two signal elements (0 and 1) then the baud rate is equal to the bit rate.As this is the case on the transputer we will mostly refer to bit rate.
- data rate - It is the number of data elements (bytes) transmitted per unit of time. Normally it is expressed in Bytes per Second. It is always smaller than the bit rate divided by 8, due to the control bits needed in each frame.
- net bit rate - (or transfer rate) will be defined by the author as 8 times the data rate. This was used to make comparisons to the values advertised.

2. **Data Transmission Basics**

a. Modes of Operation

- parallel transfer mode: when multiple wires are used between the two equipments , each one of them carrying one bit of the frame.
- bit serial transmission: when only one wire is used to send the frame, one bit after the other.

b. Communication Modes

- simplex : when data is being transmitted in one direction only.
- half duplex: when data is being transmitted in both directions but alternately (switching between transmit and receive mode is necessary).
- duplex - (or full duplex) when data is being exchanged in both directions simultaneously.

c. Transmission Modes

- Asynchronous Transmission - when the receiver and transmitter clocks are independent. Each frame received reinitializes the clock, as the start bit is received. It is used when the rate at which characters are generated is indetermined and hence the transmission line can be idle for long periods in between each transmitted character.

- Synchronous Transmission - When receiver and transmitter clock are dependent and information is packed in long streams of characters instead of byte by byte. Use special synchronizing bytes before each block.

Most of the information contained in this section was taken from [Ref. 19], and it is just included here to make the reading smoother.

B. THE TRANSPUTER LINKS

> "The transputer architecture simplifies system design by using point to point communication links. Every member of the transputer family has one or more standard links, each of which can be connected to a link of some other component. This allows transputer networks of arbitrary size and topology to be constructed."

This quotation extracted from [Ref. 7: p.6], gives us a macro sense of what the link is and how it can be beneficial for the programmer. Following the terms described in the previous section, we can say that the transputer links are serial, full duplex, asynchronous communication devices that have a bit rate of 10 mbits/sec or 20 mbits/sec (when available). They provide synchronization between communicating processes on a transputer network.

To provide the reader with a better understanding, the following includes some details about the links, extracted from [Ref. 7: p.7]:

- Each physical link provides two Occam channels, one in each direction(input and output). The T-414 has four(4) links, so we have 8 physical channels for programming purposes in each transputer.
- Communication via any link may occur concurrently with communication on all other links and with program execution.
- Synchronization of processes at each end of a link is automatic and requires no explicit programming. This is one of the important features one can use with the transputer. The links are the concurrency tools and are very easy to program by using the Occam channels.
- The information is transmitted on the link in the format depicted by Figure 2.1 , where the two beginning "1" are start bits and the ending "0" is the stop bit.
- After transmitting a data frame (one byte), the sending transputer waits for an acknowledge (ACK) from the receiving transputer, signifying that the byte was received and it the link is ready to receive another byte. If the ACK is not received the communications on that link will stop.

It is still worth mentioning one of the questions we had about how they work:

- "How could a process waiting for communication waste no cpu cycles?

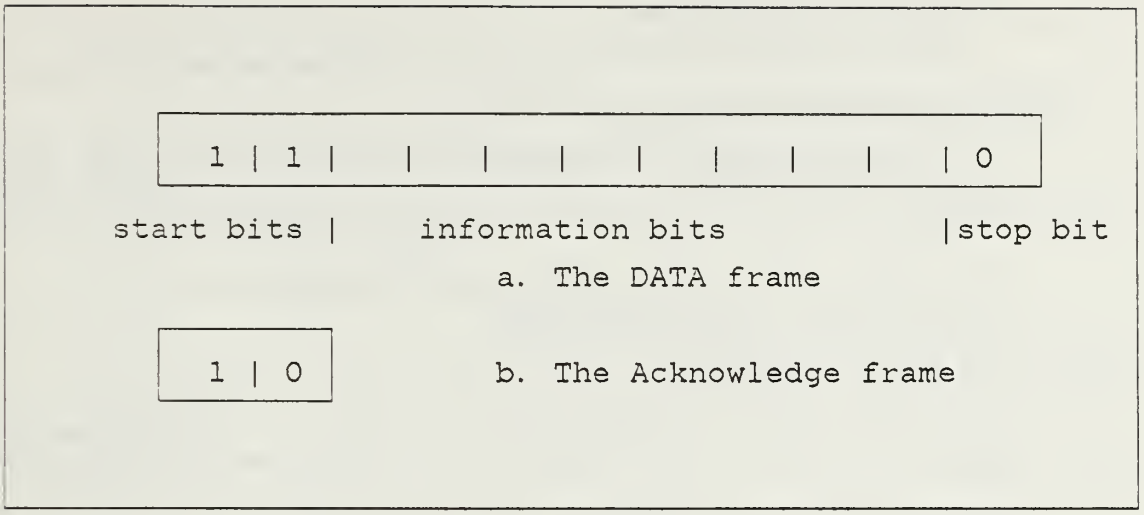


Figure 2.1 The Data and Acknowledge Frames.

The information we have got verbally from Mr. Neil Mitchell from INMOS office in Santa Clara was that the links have a 1 byte buffer inside it. When a process has to transmit, the first byte of the message is, in fact transmitted and it is received by the link on the receiving transputer, and stored in this buffer. Two situations may occur then:

1 If that receiving link is already waiting for an input, the acknowledgement is sent right away to the transmitter by the other channel, and this is all the transmitter needs to follow on with the message.

2 If that receiving link is not waiting for any input yet, the acknowledgement is not sent. What happens in the sending transputer is that, as the ACK does not arrive, the process is placed on the wait queue, and a pointer to that process is placed in the respective channel memory location (each channel has its own) until the ACK arrives. When this happens, the process is awakened and the message is then transmitted.

C. EXPECTED RESULTS

Assuming we have a bit rate of 10 mbits/sec and the frames have no delay between them, two situations should be considered:

- One Channel Transmitting - when only one of the channels is being used for transmission (or reception) of messages at a time, and so the other channel is free to bring the ACK frames.
- Both Channels Transmitting/Receiving - when we have message passing in both channels at the same time and so the ACK for a received frame is piggy-backed (appended to the end of the frame) [Ref. 19: p. 129], to the next transmitting frame.

1. **One Channel Transmitting**

In this case, there is no ACK sharing time with the frame on the channel and we will get the maximum rate possible as follows:

- Net bit rate = (8/11) * 10 (mbits/sec) = 7.27 mbits/sec or 7,273 kbits/sec. Where 8 is the number of information bits and 11 the total number of bits in a frame.
- Data rate = 7.27 / 8 = 0.91 mbytes/sec or 909 kbytes/sec.

2. **Both Channels Transmitting/Receiving**

In this case we will have:

- Net bit rate= (8/13) * 10 (mbits/sec)= 6.15 mbits/sec or 6,154 kbits/sec; where 13 stands for the 11 frame bits plus 2 ACK bits that are now sharing the link also.
- Data rate = 6.15 / 8 = 0.77 mbytes/sec or 769 kbytes/sec.

The results are summarized for 10 and 20 mbits/sec rates in Table 4 .

TABLE 4

EXPECTED MAXIMUM TRANSFER RATES ON THE TRANSPUTER LINKS

link bit rate	10	20	mbits/sec
One channel	7,273	14,545	kbits/sec
Both channels	6,154	12,308	kbits/sec

The reason for mentioning the values in kbits/sec is due to the non-availability of floating point and this way, to get some precision, we needed to use this unit in all performance measurements during the evaluation.

It is worth mentioning that these values were expected for either one single channel, or the eight channels operating in parallel because the memory is multi-ported and permits access to each one of the links and the processor in an interleaved mode. [Ref. 7: section 2, p.1]. It was also expected that these rates should not be affected by another process using the Central Process Unit (CPU) for calculations and memory accesses at the same time, for the same reasons mentioned above.

D. RESEARCH QUESTIONS

From the above, some research questions could be devised as follows:

1 Does a link transmit at 10mbits/sec and 20 mbits/sec transfer rate?

2 Is the ACK really transmitted as soon as the receiver channel receives the first bit of the data packet?

3 Is the communication between the transputers really occurring in parallel?

4 What is the effect of message lengths on the link transfer rates?

5 What is the mutual effect on the link transfer rates, of more links operating in parallel in the same transputer?

6 Can the CPU work in parallel with all the links?

7 What is the effect of a communication independent process, running on the CPU, over the transfer rates obtained in a link by another process, in this transputer?

8 What is the effect of the communications, over the process that is being executed in the CPU?

9 Does the Transputer achieve linear performance improvement?

10 What happens when a process is time sliced in the middle of a communication by physical link? Does the link stay blocked?

Questions 1 through 6 will be discussed in Chapter III, questions 6 to 9 in Chapter IV. Question 10 is still pending and is left for further research.

E. PERFORMANCE MEASUREMENT ISSUES

As mentioned in the paper by Cellary [Ref. 20], there are five methods for computer network measurements, depending on the approach used for data gathering. They are:

- Standard User Method,
- Reference User Method,
- Software Monitoring Method (Programs),
- Hardware Monitoring Method (Probed Equipments), and
- Hybrid Monitoring Method (A mix of the two above).

In this thesis both Software and Hardware monitoring methods were used for the following reasons:

- The hardware monitors are more reliable than the software monitors.
- For statistics purposes and for large amount of data, some times it is impossible to obtain, using hardware measurements, the same amount of information that can be collected by software programs, in a same period of time.

This way, we used hardware monitors to confirm preliminary results obtained by software and after validating them, a massive collection of data was gathered to permit and back up the conclusions reached.

1. Hardware Methods

Two approaches were used:

- by using a Oscilloscope to monitor 1 or 2 channels of a link at the same time.
- by using a Logic Analyzer to monitor 4 and 8 channels (in 4 different links) of the same transputer.

a. Using the Oscilloscope

The idea of using the Oscilloscope was to identify on the screen a known pattern of bits in continuous transmission, and also to obtain an approximation of the bit rate. Also by observing subsequent frames, try to estimate the data rate and the interval between frames. Another observation that could be made, as seen in the following Chapter and also documented by using Polaroid photographs, is the relative position of the Acknowledge (ACK) frame, in reference to the transmitted frame, in the second oscilloscope channel.

The equipment used was the Tektronix 364 Storage Oscilloscope and the camera was the Hewlett Packard HP-24A.

b. Using the Logic State Analyzer

The idea of using the Logic Analyzer was to monitor several channels of a same transputer and really see if there were bits been transmited at the same time, in some or all of the channels. Our Logic Analyzer has the capability to monitor 32 channels and store 250 subsequent bits in each in each channel after triggered.

As all channels are asynchronous, an external clock was necessary and so a Pulse Generator was used to provide this clock. To help in getting a more precise clock a Digital Counter was also used to sample it. The equipments used were:

- Logic State Analyzer Mod. 532 with Analyzer Probe Model 51A.
- 20 mhz Function/Pulse Generator Wavetek Model 145.
- Measuring System Hewlett Packard model HP-5300A.

One problem arose from this:

- The maximum external clock frequency acceptable by our logic analyzer was 12 mhz and as recommended by Nyquist relation, we should have a sampling frequency at least the double of the sampled signal (Normally 16 times is used) [Ref. 19: p. 15].

In our case, the sampled signal was supposedly at 10 mhz and so a minimum clock of 20 mhz should be used. As the Logic Analyzer did not permit that, we used a 10mhz pulse instead as clock, and, by trial and error varying the clock frequency and pulse width, after numerous tentatives we obtained some representative

results that are presented in the following chapter. It is good to mention that we did not even try to monitor the links running at 20 Mhz for the same reason.

c. Test Points

To monitor the links activity, a homemade monitoring bridge that was able to connect up to eight channels was used and, with it, we had the ability to monitor the four links of a transputer.

2. Software Methods

With this respect, several programs were made at first to compare the rates obtained in hardware with the ones in software, and for the final report on the links performance, a complete Link Evaluation Program was designed, to handle all possible cases of constructs to communicate, several kinds of channel parallelism and two different cases of CPU load, concurrently with the communications. The output of this program was a table of values that was used to generate some graphics using the EASYPLOT system at the IBM 3033. Appendix E presents a listing of the evaluation program with the Occam library used. The terminal driver is the one provided by INMOS, with the Keyboard and Screen references made using the first letter in uppercase, and therefore is not included.

The library.occ is a collection of previous existing procedures, some generated by the manufacturers and some made originally for the OPS System by previous workers, updated to be used on the TDS, plus additional procedures for i/o and utilities written by Cordeiro and myself. They can be browsed on Appendix E, inside the program listing.

To observe the effect of multiple transputer execution of the same program, a series of versions of Program TEST LINEARITY were made and the 17 transputer version is listed in Appendix F.

All programs above used basically the same three tools:

- - The TIME channel provided by the compiler and Occam to read the internal transputer clock in ticks. Table 5 summarize them.
- - the tick.to.time procedure used to convert time from ticks into hours, minutes,seconds and milliseconds. It receives as input parameters the "startime" (in ticks), the "endtime" (in ticks) and the transputer type, and outputs to the screen the elapsed time in hours, minutes, seconds and milliseconds, for the specified transputer. This routine is listed in Appendix F.
- - the transfer.rate procedure similar to the previous one but which computes the transfer rate measured in the channel observed. It receives as parameters "startime", "endtime", "transputer type nr.", and the "size of the message"

TABLE 5
THE DIFFERENT TICK VALUES

```
T-414 12mhz ----------> 1 tick = 1.6 micro-seconds
T-414 15mhz (high) ---> 1 tick = 1   micro-second
T-414 15mhz  (low) ---> 1 tick = 64  micro-seconds
VAX/VMS --------------> 1 tick = 100 nano-seconds
```

transmitted and outputs the transfer rate through the variable "rate". This routine is listed in Appendix E.

III. THE EVALUATION STARTS

A. INTRODUCTION

In this chapter we start to address the research questions related to the evaluation, as listed in Section D of Chapter II.

Section B describes how we verified that the bit rate is indeed, 10 mbits/sec or 20 mbits/sec. It also shows the maximum values achieved for the net bit rate (transfer rate), for the various construct types.

Section C shows the configuration used and demonstrates that the transfers in different links occur in parallel, eventually in all 8 channels of the 4 links.

Section D describes the message size, and the channel parallelism effects on the transfer rates for the various constructs.

1. The Available Constructs

Occam permits us to use several different primitives and procedures for communications between processes. The first to be mentioned are the **input** and **output,** already explained in Chapter I. We used them in two modes:

- transmitting bytes (characters), or
- transmitting words (integers).

```
BYTE.SLICE.OUTPUT (chanid, buffername, initbyte, blocksize)
where:
- chanid     - the channel name where the communication will occur
- buffername - the name of the array of variables
- initbyte   - the array index of the first byte to be transferred
- blocksize  - the number of bytes to be transferred
```

Figure 3.1 The BYTE SLICE OUTPUT Procedure Call.

The third mode is the BYTE SLICE INPUT and BYTE SLICE OUTPUT procedure. These procedures are microcoded subroutines that provide a block transfer of bytes. Figure 3.1 shows the procedure call and an explanation of the parameters

[Ref. 14: section 4]. These procedures cannot be used when doing programs for the OPS. The advantage they bring us, is a better performance, but when using OPS we are not concerned about it.

The last mode is the WORD SLICE INPUT and WORD SLICE OUTPUT procedure, also microcoded, that provide block transfer of words. As just mentioned above, the procedures showed to be much faster than the **input/output** primitives, but with similar performance to the BYTE SLICE procedures.

2. **Considerations About Memory Management**

As we have a machine with internal and external memory with different performances and address capabilities, this was a major concern, as far as performance could, and in fact is, undoubtedly affected. The documentation is not clear enough to permit us to assure how this is handled by the processor, in the b001 and b003 boards. We tried to check the addresses mentioned in [Ref. 7: section 2, pp. 5,7], but we were not able to verify that.

What can be said, though, is that it looks like the memory (internal plus external) on the B001 transputer board is divided into four memory banks, each one of them beginning at addresses 0, 16k, 32k, and 48k , and the data and programs are mapped evenly over these four banks. We reached this conclusion after browsing several listings of the memory contents obtained from the transputer in the B001 board. through a "dump" routine designed and implemented by M. Cordeiro, also part of the LIBRARY.OCC, included in Appendix F.

In our evaluation program outputs, we tried to observe any noticeable effects that could be explained by a fast or slowest memory access, but the evidences were not strong enough, as it will be mentioned further on. As a curiosity, we measured the time to initialize four arrays of 15,000 bytes each in the B001 memory and we have got 133 msec! We assumed that programs smaller than 2k bytes long, will be loaded entirely into internal memory, but we could not prove it and this is left and strongly recommended for further research.

B. A CLOSE LOOK ON THE BIT RATE

The evaluation started trying to answer research question 1 that is transcribed here:

- "Do the links transmit (and receive) data at 10 and 20 mbits/sec transfer rates?"

1. First Software Results

To find that out, simple programs were made to transmit and receive long messages (arrays) through the physical links. The transfer rate was obtained by dividing the number of bits transmitted by the time spent on the transmission. A flag was used (single byte) from receiver to transmitter to assure the transmitter would only transmit when the receiver was ready. This way, we would be timing the best possible case with the best possible accuracy. The basic program code used for the transmitter and receiver is in Figure 3.2 The BYTE SLICE was the construct used, because from the very first tests it proved to be the fastest, even for one byte being transmitted.

The configuration used for that was as simple as it could be. Two transputers connected by a link hosting one procedure transmitter (TR.1) and one procedure receiver (TR.2). Figure 3.3 depicts that.

```
SEQ
   chan1 ? flag       --- flag is received
   TIME ? startime    --- time is stored in var startime
   --- transfer begins
   BYTE.SLICE.OUTPUT (chan2, buffername, 1, block.size)
   --- transfer ends
   TIME ? endtime     --- time is stored in var endtime
   --- call to procedure transfer rate outputs the rate.
   transfer.rate (startime, endtime, transputer.type,
                  blocksize, rate)
   a) Transmitter
*****************************************************
SEQ
   chan1 ! char  --- flag is sent to transmitter
   BYTE.SLICE.INPUT (chan2, buffername, 1, block.size)
   b) Receiver
```

Figure 3.2 Basic Code for Transmitter and Receiver.

The block size used was 15,000 bytes, in order to avoid possible dragging effects of small messages. The results obtained were around 3,800 kbits/sec with an execution time of 31.5 msec, average. As we can notice from table 4, in Chapter III, it was almost half of the expected value of 7,273 kbits/sec. Why? The monitoring of the channels with the oscilloscope answered this question.

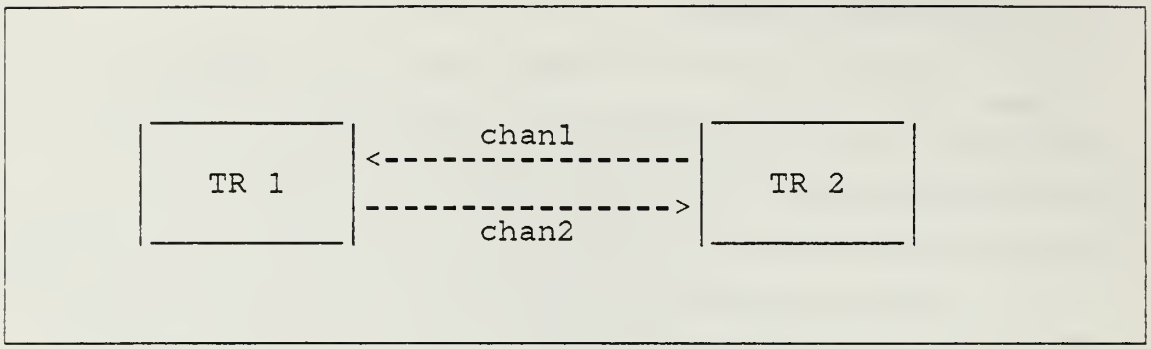


Figure 3.3 Configuration for Initial Tests.

a. Links at 20 mbits/sec

With the links switched to 20 mbits/sec, we could only have communications between transputers located on B003 boards, so, although the code was practically the same, the configuration had to be slightly different. Figure 3.4 shows us how it was.

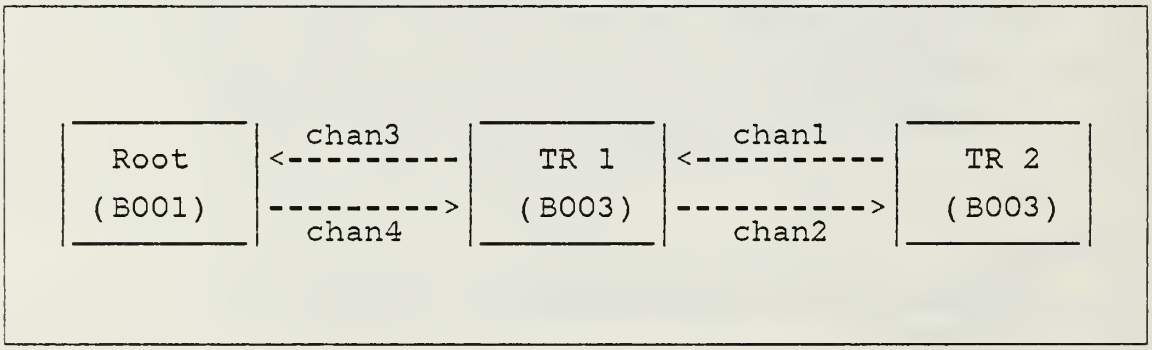


Figure 3.4 Configuration for Measuring Links at 20 mbits/sec Bit Rate.

The results obtained for block sizes of 15,000 bytes using also the BYTE SLICE construct, where of the order of 6,000 to 6,100 kbits/sec, again very small, if we compare them with the expected of 14,545 kbits/sec.

2. **Using the Oscilloscope**

Another simple program that made a continuous transfer on the link, made it possible to observe the frame transmitted and estimate the rate on the oscilloscope screen. The message transmitted, using BYTE SLICE, was a sequence of TRUES. The TRUE, in Occam, is a sequence of 8 binary 1's and so the frame was as Figure 3.5 shows.

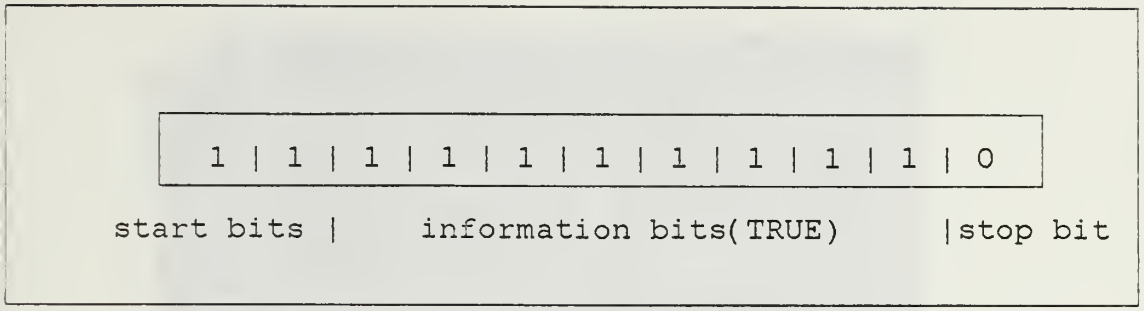


Figure 3.5 Frame Transmitted for Oscilloscope Observations.

The basic code used is depicted in Figure 3.6 . There is no time sampling or flags to avoid any side effect on the oscilloscope screen. Figure 3.7 shows the picture of a frame like the one on figure 3.5 followed by an acknowledge (both appear on the same trace due to vertical mode ADD used on the oscilloscope. All the oscilloscope settings are also mentioned below the picture.

```
WHILE TRUE
  BYTE.SLICE.OUTPUT (chan2, buffer1, 1, block.size)
a) code on the transmiter
   ********************************************************
WHILE TRUE
  BYTE.SLICE.INPUT (chan2, buffer2, 1, block.size)
b) code on the receiver
```

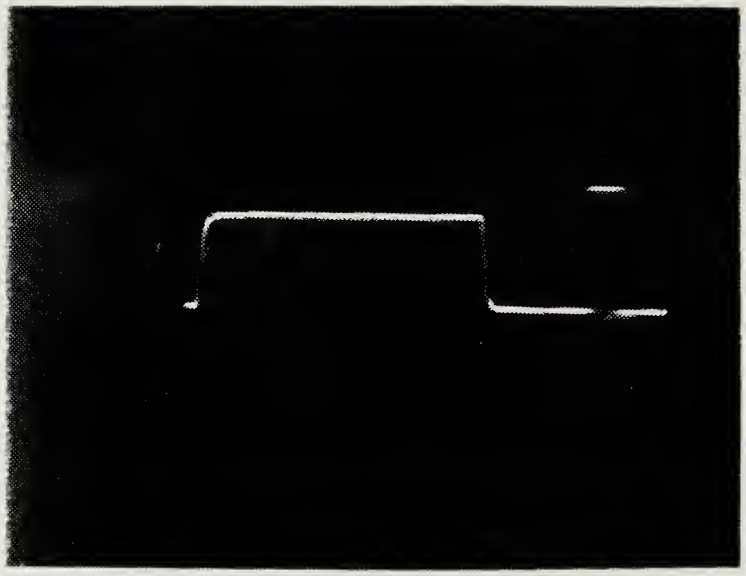
Figure 3.6 Example Code for Oscilloscope Observations.

Note from Figure 3.7 that the 10 "ones" of the frame occupy 5 divisions. This sums up to 1 microsecond. So we have one bit per 0.1 microsecond and this implies a bit rate of 10 mbits/sec (gross).

Conclusion 1

The bit rate is in fact 10 mbits/sec,

if we consider only one frame.



```
Oscilloscope Settings:
    channel 1     --> shows the transmitted frame
    channel 2     --> shows the acknowledge
    time scale    --> 0.2 microsec / division
    voltage scale -> 2 volts / division
    vertical mode -> ADD
    trigger source-> channel 1
```

Figure 3.7 Picture of One Frame at 10mbits/sec Rate.

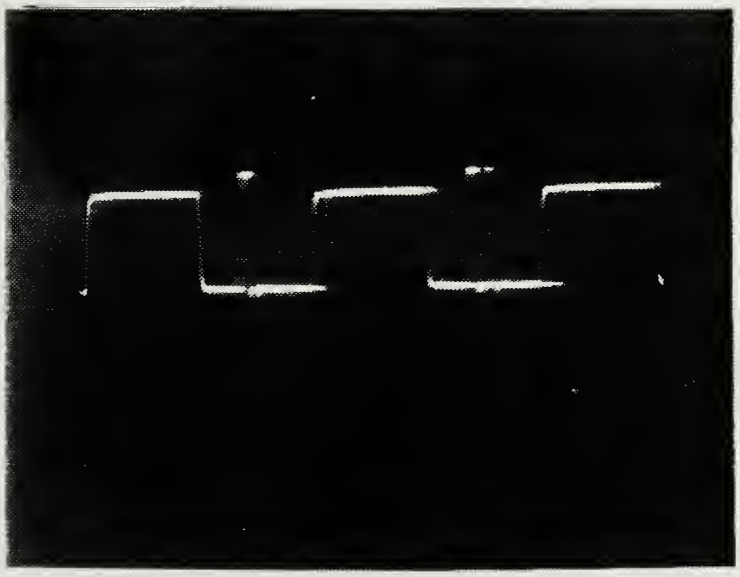
The Acknowledge appears enlarged due to the lack of synchronization between both channels and the trigger source to be oscilloscope channel 1. We can't take precise measurements, but we can estimate the best and worst cases:

- best case - The ACK pulse is beginning at the trailing edge (leftmost) of the ACK frame. This will give us a distance of approximately 200 nsec between the last bit of the data frame and the acknowledge frame (remember that there is a "zero" bit after the last "one").
- worst case - The ACK ends at the leading edge (rightmost) of the ACK pulse. This will give us a distance of approximately 300 nsec instead.

Conclusion 2

The ACK frame leaves the receiver 200 to 300 nsec

after the transmitted frame arrived!



Oscilloscope Settings:
channel 1 --> shows the transmitted frame
channel 2 --> shows the acknowledge
time scale --> 0.5 microsec / division
voltage scale -> 2 volts / division
vertical mode -> ADD
trigger source-> channel 1

Figure 3.8 Three Data Frames at 10mbits/sec Rate.

Increasing the time scale of the oscilloscope to 0.5 microseconds, we could observe more frames and acknowledges as shown in Figure 3.8, and from this picture, using the same best and worst case approach, we could estimate that the distance between the ACK and the following frame (center) is between 500 and 600 nsec. We could also notice that the distance between consecutive data frames is between 900 and 1000 nanoseconds.

So, estimating the transfer rate from the picture, assuming all frames will keep at least this space between them, we got:

- best case- for each 1100 nsec information we have 900 of line inactive. If we multiply this ratio by the expected transfer rate of 7,273 kbits/sec, we get

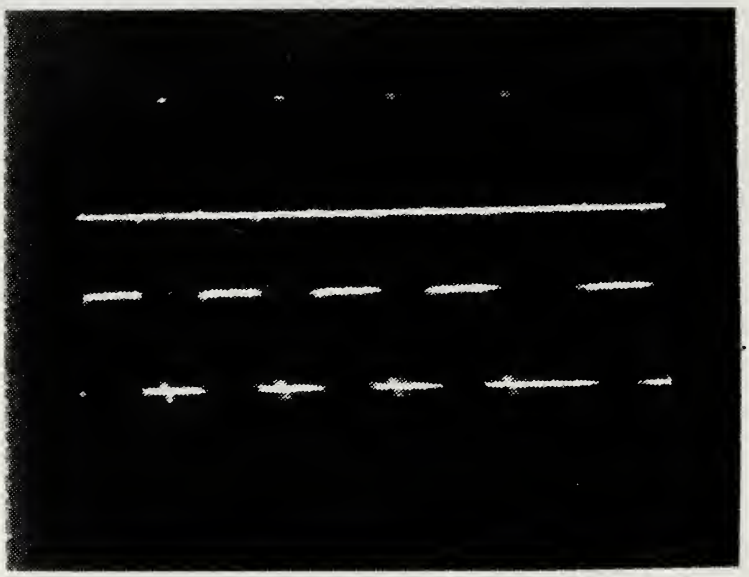
  (1100 / 2000) x 7273 = 4000 kbits/sec

- worst case - then 1000 nsec of line inactive would bring us

  (1100 / 2100) x 7273 = 3809 kbits/sec

As we can see the hardware results were confirming the previously obtained software results.

Another fact to add is that, during our observations, the frames were not always equally spaced as shown in Figure 3.8 In fact, this figure shows the most equally spaced results we ever obtained. Figure 3.9, in which the time scale was once more increased, to 1 microsec per division, we can note that the fifth frame in the channel at the bottom is more spaced than the four previous ones. In this picture the ALT vertical mode was used to permit us to see the ACK on the upper trace. Note the regularity which the acknowledge appears 200 to 300 nsec after the received frame.

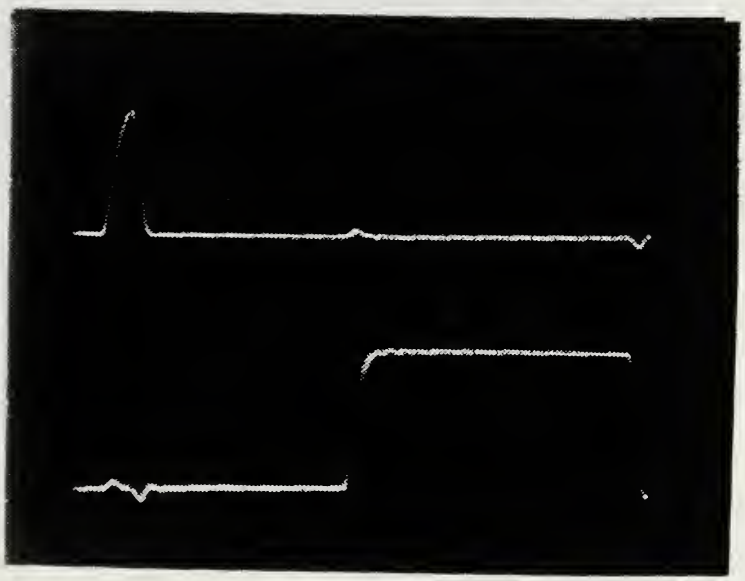


Oscilloscope Settings:
channel 1 --> shows the transmitted frame
channel 2 --> shows the acknowledge (upper)
time scale --> 1.0 microsec / division
voltage scale -> 2 volts / division
vertical mode -> ALT
trigger source-> channel 1(bottom)

Figure 3.9 Five Frames Observed at 10 mbits/sec Rate.

a. Switching the Rate to 20 mbits/sec

Similar observations were made for the links operating at 20 mbit/sec rate and Figure 3.10 that was taken with time scale 0.1 microsec per division shows the same 10 "ones" of Figure 3.7 in approximately 0.5 microsecond, that is half of the time that was obtained there. The ACK now is in oscilloscope channel 1 and is the trigger source (this is the reason it is now well defined).



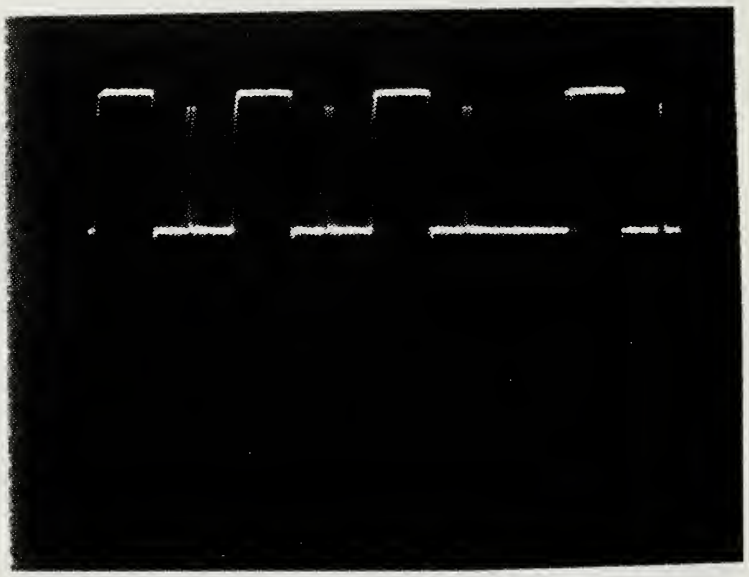
```
Oscilloscope Settings:
    channel 1 --> shows the acknowledge (previous)
    channel 2 --> shows the transmitted frame(next)
    time scale --> 0.1 microsec / division
    voltage scale -> 2 volts / division
    vertical mode -> ALT
    trigger source-> channel 1 (upper)
```

Figure 3.10 One Frame and the ACK at 20 mbits/sec Rate.

Note the time delay between the ACK (upper trace) and the following frame (lower trace) that was measured as about 400 nsec.

Figure 3.11, taken with time scale 0.5 microseconds per division shows us a series of "TRUE" frames at 20 mbits/sec rate and the ACKframes in the same trace. We could estimate the percent of time the link is actively transmitting as around 40%

of the total time approximately. If we take 40% of the predicted rate of 14,545 we get 5,818 kbits sec. Comparing this with the software obtained value of 6,100 kbits/sec, we can conclude that the value is reasonable enough for an explanation of the software results.



Oscilloscope Settings:
channel 1 --> shows the acknowledge
channel 2 --> shows the transmitted frame
time scale --> 0.5 microsec / division
voltage scale -> 2 volts / division
vertical mode -> ADD
trigger source-> channel 2

Figure 3.11 Four Frames and ACK at 20 mbits/sec Rate.

3. **Comparison Between the Constructs**

In this subsection we would like to include the maximum values of the transfer rate obtained, ever for the various constructs. They are summarized on Table 6 and were obtained using different programs, with different message sizes and so they are not mentioned. It is interesting to note that the input operation has a slight tendency to be quicker than the output, which is not true. This occurs because of the flag positioning, which will slightly affect the rate, but the rate should be considered as the

same. What can be mentioned, however, is that [Ref. 7: section 2, pp.26,27], shows us an expected performance summary and there the **input** primitive is rated as using 26.5 processor cycles while the **output** would take 26 cycles, and this is not much of a difference. This same reference still mentions that the values are not definitive and may suffer changes as more information is collected.

TABLE 6
MAXIMUM TRANSFER RATES OBTAINED (KBITS/SEC)

	input/output primitives		BYTE SLICE procedure	WORD SLICE procedure
	bytes	words		
output	595	2412	3880	3669
input	631	2855	3804	3786

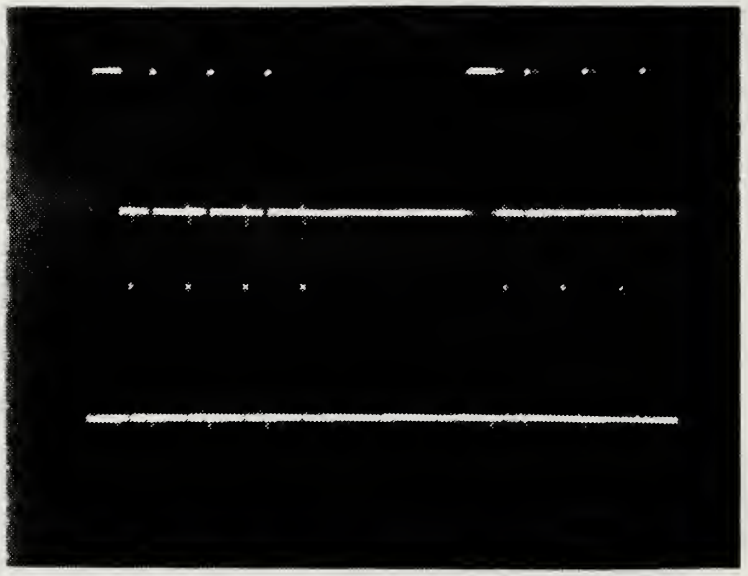
Browsing the figures on Table 6, one question comes up at once:

" Why is the transmission using the input and output primitives, so much slower in comparison to the built-in procedures?"

For the byte transmission case, using the primitives, if we look at Figure 3.12, we will see how an array of "TRUES" is transmitted through the link, at 10 mbits/sec selected bit rate. The information seems to be stored one byte per word and this way, for each "TRUE" byte, three empty frames follows. Note that the frames carry only the start bits (two "ones"). The time between frames containing information, measured at the lab was 13 microseconds.

For the word (integer) transmission case, if we browse Figure 3.13, we see a similar pattern to Figure 3.12 but with the difference that all frames are effectively carrying information bits. The information used to ease the observation was maxint, which is, for our 32 bit machine 2,147,483,647 decimal or " 7FFF " hexadecimal.[9] The elapsed time measured at the lab between the acknowledge of the last byte of the first word and the first byte of the second word was around 5 microseconds. By doing same calculations done for the BYTE SLICE case one will conclude that the maximum values obtained are in accordance with the observations on the oscilloscope.

[9]The transputer T 414 uses signed integers in the range - 2,147,483,648 to 2,147,483,647 decimal or 8000 to 7FFF hexadecimal, respectively [Ref. 7: section 2, p. 2].

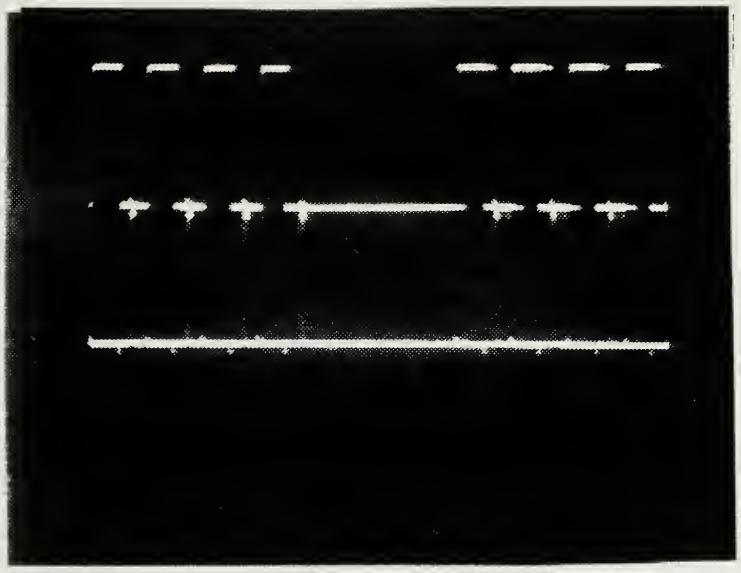


Oscilloscope Settings:
channel 1 --> shows transmitted frame (upper)
channel 2 --> shows the acknowledge (lower)
time scale --> 2.0 microsec / division
voltage scale -> 2 volts / division
vertical mode -> ALT
trigger source-> channel 1

Figure 3.12 TRUES Transmitted Using the Input/Output Primitives.

As a conclusion of this section, we could prove that the software measurements and the procedures used to calculate the transfer rate were producing reasonable values, that agreed with those observed on the oscilloscope. The reason we had not obtained the expected transfer rates was because the link is not continuously active as the literature led us to believe, and there is a considerable delay between the receipt of a frame and the departure of the corresponding ACK. Also, after the ACK is received by the transmitter, there is another delay to transmit the next frame.[10]

[10]In fact during the Occam User Group meeting already mentioned, in Santa Clara, CA, Mr. Martin Booth from INMOS office at Santa Clara said that the data rate we should really expect on the links was 450 kbytes/sec, what agrees with our results (450 x 8 = 3800).



Oscilloscope Settings:
channel 1 --> shows transmitted frames (upper)
channel 2 --> shows the acknowledge (lower)
time scale --> 2.0 microsec / division
voltage scale -> 2 volts / division
vertical mode -> ALT
trigger source-> channel 1

Figure 3.13 Maxint Transmitted Using the Input/Output Primitives.

Although, it is expected, that the new transputer version, the T 800, will solve this problem by permitting the acknowledge leave the receptor, as soon as the first bit of the frame arrives, and this way the delay would not exist, or at least be smaller [Ref. 21].

C. OBSERVING PARALLEL ACTIVITY ON THE LINKS

1. Using Software

To observe the links working at the same time, we needed to build a different configuration. As we have 4 links per transputer, we needed at least 5 transputers to make all links work in parallel at the highest possible rate. The configuration used is depicted in Figure 3.14 .

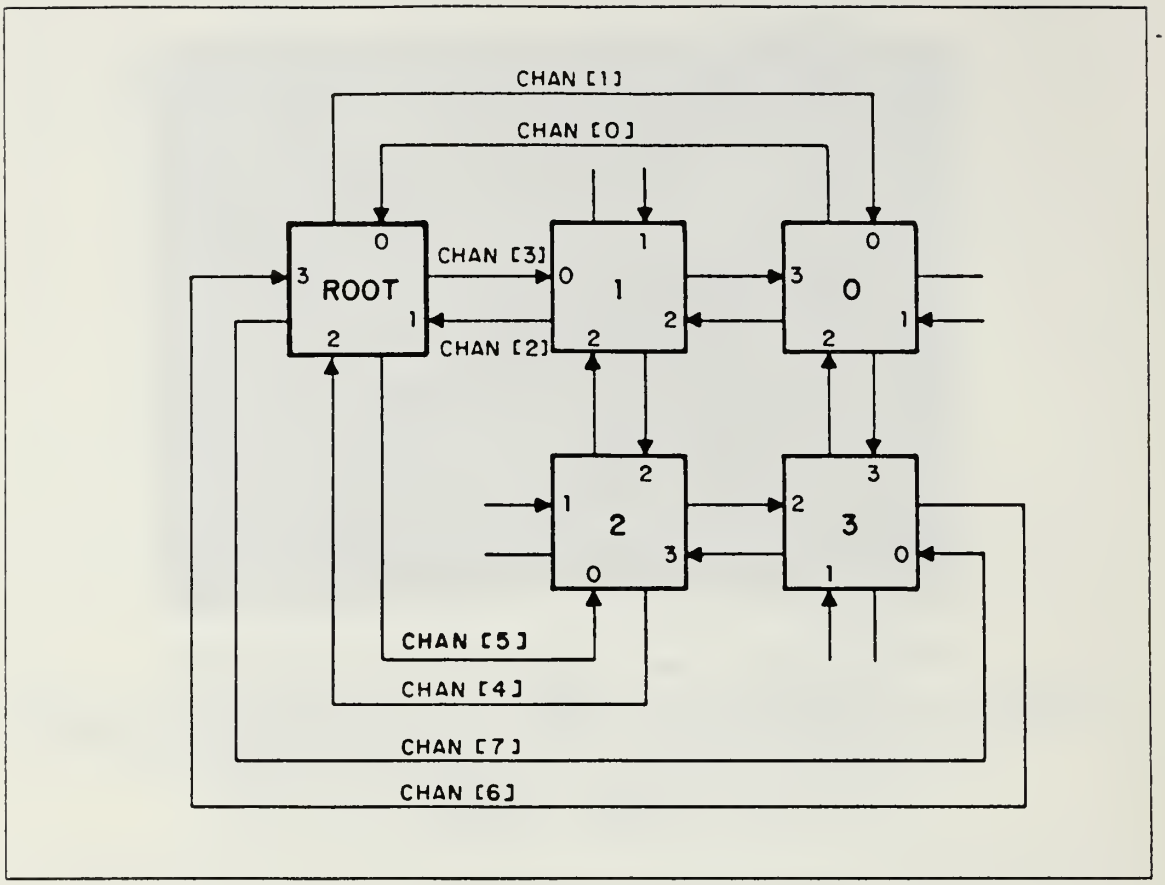


Figure 3.14 Configuration to Observe the Four Links Operating in Parallel.

In processor root we placed a procedure that was able to transmit and/or receive in parallel to/from the four transputers 0,1,2 and 3. Figure 3.15 shows the Occam code used to do that.

Using Figure 3.15 as a template, one can implement a similar code to transmit in 2 or 3 channels in parallel just by removing the unwanted BYTE SLICE procedure calls. Note that the channels mentioned on Figure 3.15 are in accordance with the ones on Figure 3.14 (1,3,5,7 are output channels for transputer root). On the other hand one may think of using the input channels at the same time, but this will be addressed in next section.

The receivers in their turn have a simpler code than the transmitter, because each one of them is only communicating with transputer root. Figure 3.16 shows it.

If one tries to map the channels of Figure 3.16 to the configuration, one will notice that there is no such a channel **in** or **out** on Figure 3.14 and to clarify that

```
PROC transmitter (CHAN chan0, chan2, chan4, chan6,
                            chan1, chan3, chan5, chan7)=
  ... declarations
  SEQ
    ... buffers initializations
    PAR
      chan0 ? flag0      --- flags are received from each
      chan2 ? flag1      --- of the receiving transputers
      chan4 ? flag2      --- and only after all of them
      chan6 ? flag3      --- are ready the timer is started
    TIME ? startime
    PAR
      BYTE.SLICE.OUTPUT (chan1, buffer0, 1, block.size)
      BYTE.SLICE.OUTPUT (chan3, buffer1, 1, block.size)
      BYTE.SLICE.OUTPUT (chan5, buffer2, 1, block.size)
      BYTE.SLICE.OUTPUT (chan7, buffer3, 1, block.size)
    TIME ? endtime
    --- transfer rate calculated will be in VAR rate
    transfer.rate (startime, endtime, 1, blocksize, rate):
```

Figure 3.15 Code Used to Time Transmission Through the Four Links in Parallel.

```
PROC receiver (CHAN in, out)=
  ... declare variables
  ... initialize buffer
  SEQ
    out ! flag
    BYTE.SLICE.INPUT (in, buffer, 1, block.size):
```

Figure 3.16 Code for the Receivers.

Figure 3.17 shows how the configuration would be actually coded for these processes to be mapped and work properly.

As one may notice from Figure 3.17, the chan0 inside the procedure refers to the chan[0] on the configuration, and so on. We could think of chan[0] being the actual parameter and chan0 being the correspondent formal. This is not strictly true, because on the configuration we are only placing the procedure on the processor, not calling it, but the analogy is still valid and the names were chosen to make it easier to understand. The users and programmers may use any name for channels, and in fact we used some different ones in our implementations. The importance is to get the idea.

```
--- configuration
DEF root = 100:--- assigning a number to root
CHAN chan[8]:--- channel variables for physical channels
PLACED PAR
  PROCESSOR root
    --- placing channel names on physical channels
    PLACE chan[0] AT link0in  :
    PLACE chan[1] AT link0out :
    PLACE chan[2] AT link1in  :
    PLACE chan[3] AT link1out :
    PLACE chan[4] AT link2in  :
    PLACE chan[5] AT link2out :
    PLACE chan[6] AT link3in  :
    PLACE chan[7] AT link3out :
    --- placing the procedure to be executed on the processor
    transmitter (chan[0], chan[2], chan[4], chan[6],
                 chan[1], chan[3], chan[5], chan[7])
  PROCESSOR 0
    PLACE chan[0] AT link0out :
    PLACE chan[1] AT link0in  :
    receiver (chan[1], chan[0])
  PROCESSOR 1
    PLACE chan[2] AT link0out :
    PLACE chan[3] AT link0in  :
    receiver (chan[3], chan[2])
  PROCESSOR 2
    PLACE chan[4] AT link0out :
    PLACE chan[5] AT link0in  :
    receiver (chan[5], chan[4])
  PROCESSOR 3
    PLACE chan[6] AT link0out :
    PLACE chan[7] AT link0in  :
    receiver (chan[7], chan[6])
```

Figure 3.17 Configuration Code for the Link Evaluation Program.

Using the program described above, the results obtained for a block size of 1,500 bytes were :

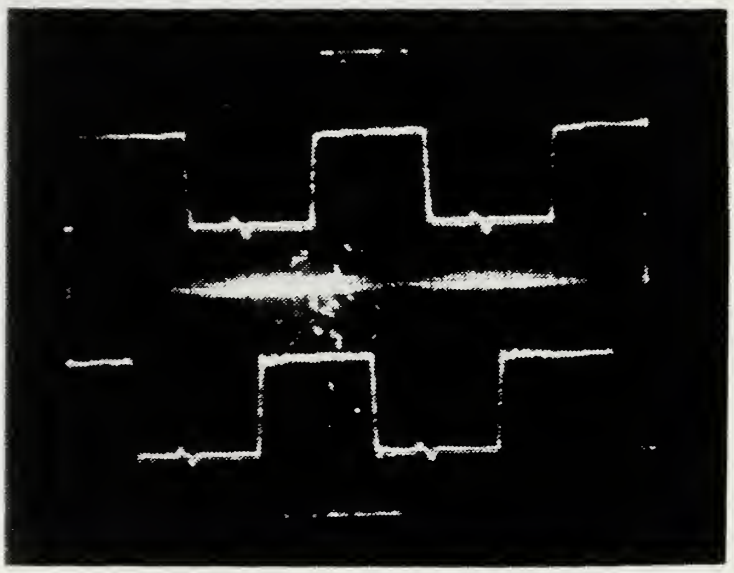
- 1 channel ..... 3670 kbits/sec
- 2 channels in parallel ..... 3670 kbits/sec(in each channel)
- 3 channels in parallel ..... 3650 kbits/sec(in each channel)
- 4 channels in parallel ..... 3630 kbits/sec(in each channel)

These results show a slight decreasing performance as more channels are in parallel, but there is nearly linear improvement in communication performance due to parallelism, because the overall data transmission jumped from 3670 to 14520 (4 x 3630)!

2. Using the Oscilloscope

As had happened with the initial observations related on the previous section, the programs used for this observation where adaptations of the ones just presented using the WHILE TRUE construct to permit continuous transmission. and taking off all timing and flags, so we will not repeat them here. As we know, the maximum we could monitor at one time, was two channels. Two observations were then made:

- two channels of different links transmitting in parallel (Figure 3.18),
- two channels of the same link transmitting in parallel (Figure 3.19).

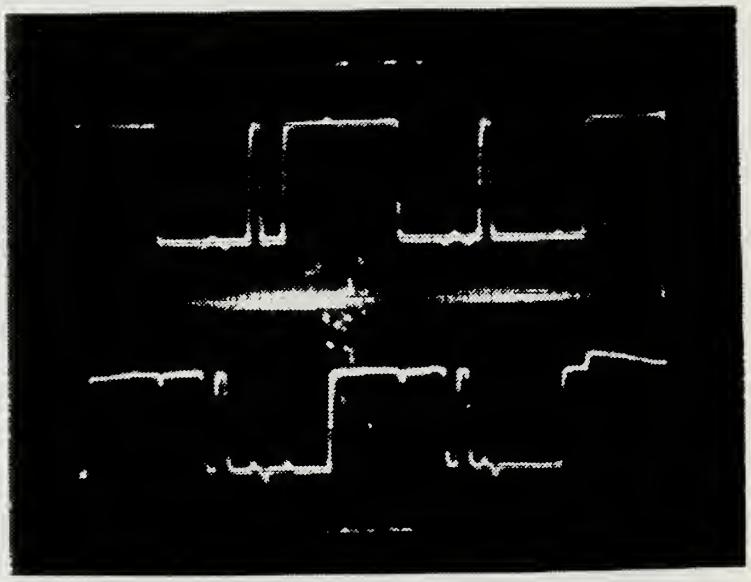


```
Oscilloscope Settings:
    channel 1 --> shows transmitted frame (chan1)
    channel 2 --> shows transmitted frame (chan3)
    time scale --> 0.5 microsec / division
    voltage scale -> 2 volts / division
    vertical mode -> ALT
    trigger source-> channel 2(lower)
    storage mode used
```

Figure 3.18 Two Channels of Different Links Transmitting at the Same Time.

In the first case, Figure 3.18 shows the same frame used on the previous section (TRUE) in two different channels and one may notice how they overlapped. It

is worth emphasizing that the interval between frames is due to the acknowledge delay explained in last section (not shown here), and the different phase between the wave forms is due only to the communication processes had begun at different absolute times. This implies that, as the links have the same speed, and as the procedures are the same, this difference of phase is kept constant as long as the processes are running. This photograph was taken in storage mode due to the fact that in normal mode the unsynchronism between the channels did not permit us to see both waveforms clearly.



Oscilloscope Settings:
channel 1 --> shows transmitted frame (chan1)
channel 2 --> shows received frame (chan0)
time scale --> 0.5 microsec / division
voltage scale -> 2 volts / division
vertical mode -> ALT
trigger source-> channel 2(lower)
storage mode used

Figure 3.19 Two Channels of the Same Link Operating at the Same Time.

In the second case, Figure 3.19 shows two channels of a same link operating at the same time. This picture was also taken in storage mode for the same reasons just mentioned. Note that at this time we can observe the acknowledges piggy-backed on

the transmitted frames. The ACK on the upper trace are sent for the frame been shown on the lower trace, just a little before in time. The reverse is valid for the lower trace' ACK. Note that at the time the picture was taken, all eight channels were operating in the same way.

3. Using the Logic Analyzer

As mentioned in Chapter III, a snapshot of parallel operations is not easy to get. Our Model 532 Logic State Analizer could store up to 250 words of 32 bits each monitored by 2 Logic Probes of 16 bits each. We used only one of the probes once monitoring 4 channels, and another time 8 channels. In the first case, monitoring 4 channels, three of them were carrying data frames transmitted by different links (the handy "TRUE"s), and the last one carrying the acknowledge of the fourth link. Figure 3.20 is a reduction of the printout obtained from a representative part of the 250 words. Remember that the links are serial communication devices and the probes are more effective when monitoring parallel buses, specially if there is a clock available on the bus (synchronous buses), and so the sequence of "ones" appear vertically on the picture. The program being used was the EVALCONTTRUE.tds, using the same configuration depicted on Figure 3.14, and the channels monitored were chan1, chan3, chan5 and chan6. The program was transmitting continuously blocks of 15000 bytes of trues by the four output channels (1,3,5,7) of transputer root. The first column is the memory position of the logic state analyzer. In the second column bits 4 and 8 (from left to right) carry respectively chan1 and chan3. The third column bits 4 and 8 again refer to chan6 and chan5, respectively.

Figure 3.21 as well shows us one representative section of the samples collected by the logic analyzer from eight channels distributed as Table 7 shows.

The "one" bits that appear in the other columns are probably cross-talk due to the probe being made of parallel wires, while the INMOS links are always in twisted pairs. It is also good to reinforce that when the links are transmitting and receiving in parallel, the acknowledge appears piggy-backed, as Figure 3.19 shows, and in the sequence of bits we can notice them very clearly in several spots.

Although, by the evidence from both the logic analizer and the oscilloscope, we are sure that the channels indeed operate in parallel, it would be more satisfying to obtain data which more closely coincides with the measurements taken by software experiments. Our suggestion is that monitoring the channels with a logic state analyzer which can handle a faster clock, would enable a more exact measurement of

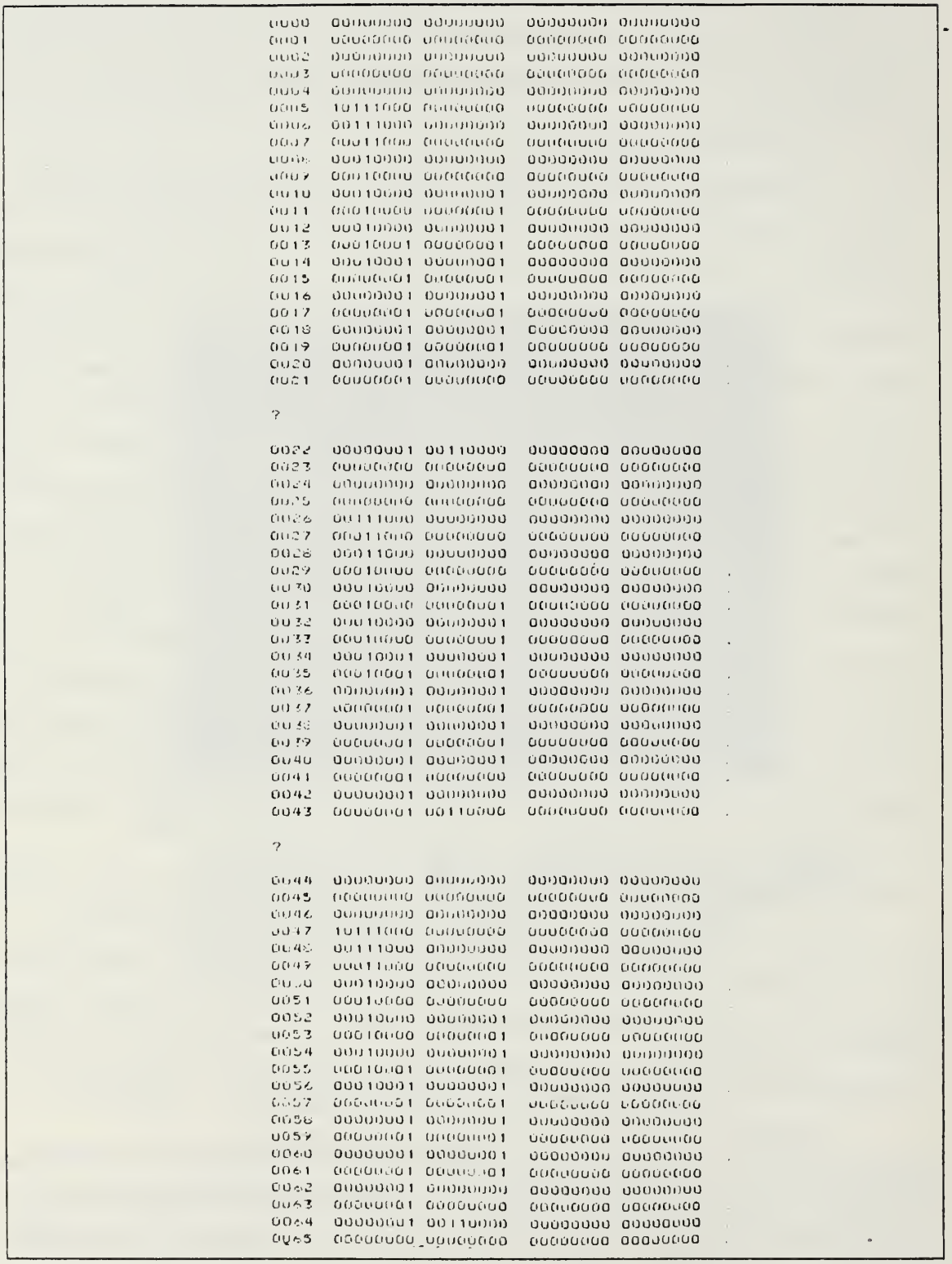
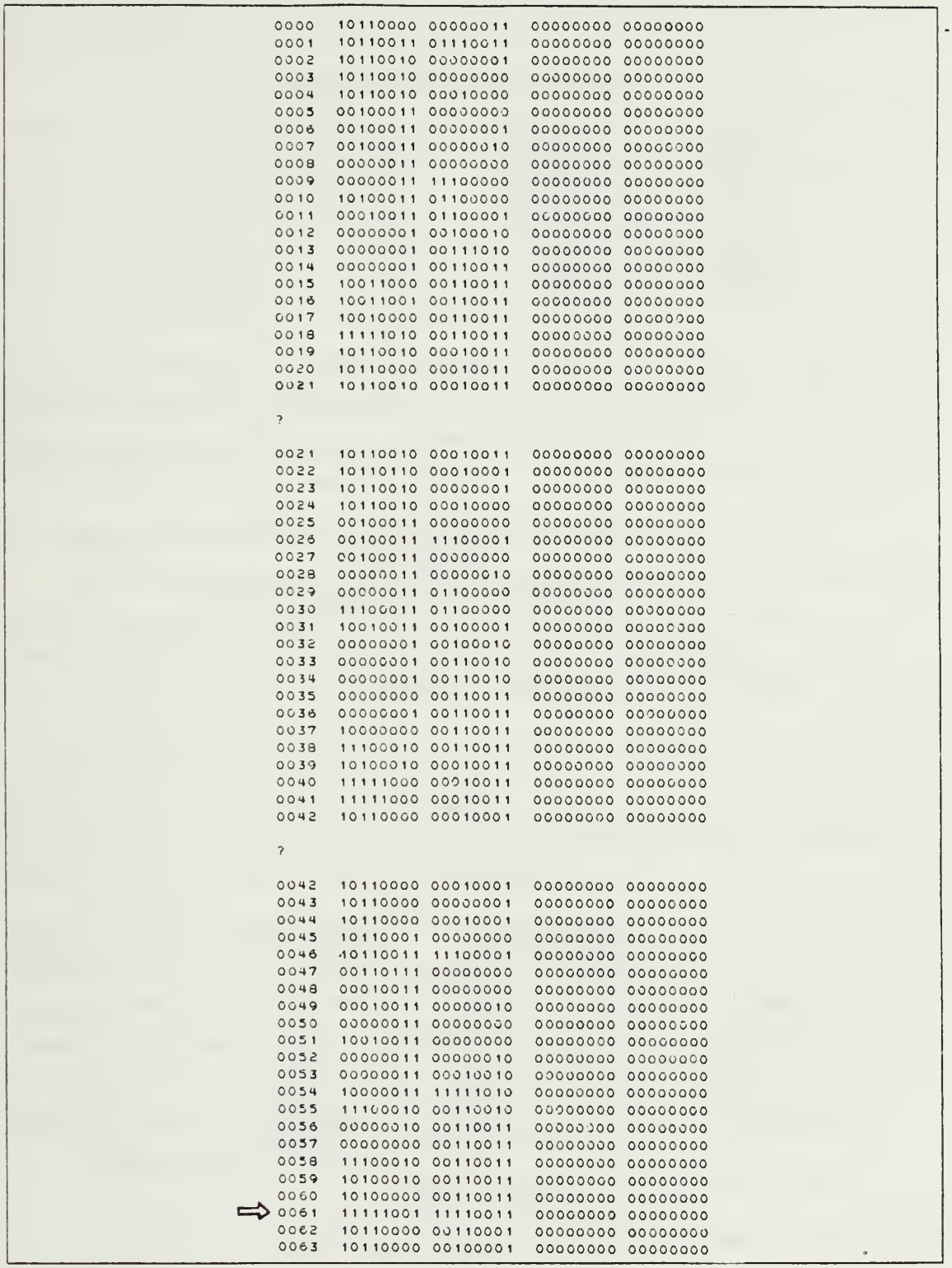


Figure 3.20 Output from the Logic Analyzer of 4 Channels in Parallel.



```
0000  10110000 00000011  00000000 00000000
0001  10110011 01110011  00000000 00000000
0002  10110010 00000001  00000000 00000000
0003  10110010 00000000  00000000 00000000
0004  10110010 00010000  00000000 00000000
0005  00100011 00000000  00000000 00000000
0006  00100011 00000001  00000000 00000000
0007  00100011 00000010  00000000 00000000
0008  00000011 00000000  00000000 00000000
0009  00000011 11100000  00000000 00000000
0010  10100011 01100000  00000000 00000000
0011  00010011 01100001  00000000 00000000
0012  00000001 00100010  00000000 00000000
0013  00000001 00111010  00000000 00000000
0014  00000001 00110011  00000000 00000000
0015  10011000 00110011  00000000 00000000
0016  10011001 00110011  00000000 00000000
0017  10010000 00110011  00000000 00000000
0018  11111010 00110011  00000000 00000000
0019  10110010 00010011  00000000 00000000
0020  10110000 00010011  00000000 00000000
0021  10110010 00010011  00000000 00000000

?

0021  10110010 00010011  00000000 00000000
0022  10110110 00010001  00000000 00000000
0023  10110010 00000001  00000000 00000000
0024  10110010 00010000  00000000 00000000
0025  00100011 00000000  00000000 00000000
0026  00100011 11100001  00000000 00000000
0027  00100011 00000000  00000000 00000000
0028  00000011 00000010  00000000 00000000
0029  00000011 01100000  00000000 00000000
0030  11100011 01100000  00000000 00000000
0031  10010011 00100001  00000000 00000000
0032  00000001 00100010  00000000 00000000
0033  00000001 00110010  00000000 00000000
0034  00000001 00110010  00000000 00000000
0035  00000000 00110011  00000000 00000000
0036  00000001 00110011  00000000 00000000
0037  10000000 00110011  00000000 00000000
0038  11100010 00110011  00000000 00000000
0039  10100010 00010011  00000000 00000000
0040  11111000 00010011  00000000 00000000
0041  11111000 00010011  00000000 00000000
0042  10110000 00010001  00000000 00000000

?

0042  10110000 00010001  00000000 00000000
0043  10110000 00000001  00000000 00000000
0044  10110000 00010001  00000000 00000000
0045  10110001 00000000  00000000 00000000
0046  10110011 11100001  00000000 00000000
0047  00110111 00000000  00000000 00000000
0048  00010011 00000000  00000000 00000000
0049  00010011 00000010  00000000 00000000
0050  00000011 00000000  00000000 00000000
0051  10010011 00000000  00000000 00000000
0052  00000011 00000010  00000000 00000000
0053  00000011 00010010  00000000 00000000
0054  10000011 11111010  00000000 00000000
0055  11100010 00110010  00000000 00000000
0056  00000010 00110011  00000000 00000000
0057  00000000 00110011  00000000 00000000
0058  11100010 00110011  00000000 00000000
0059  10100010 00110011  00000000 00000000
0060  10100000 00110011  00000000 00000000
⇨ 0061  11111001 11110011  00000000 00000000
0062  10110000 00110001  00000000 00000000
0063  10110000 00100001  00000000 00000000
```

Figure 3.21 8 Channels Monitored with the Logic Analizer.

TABLE 7

LINK MAP FOR FIGURE 3.21

channel	column	bit	probe lid
chan0 ---->	second	3	D13
chan1 ---->	second	4	D12
chan2 ---->	second	7	D9
chan3 ---->	second	8	D8
chan4 ---->	third	3	D5
chan5 ---->	third	4	D4
chan6 ---->	third	7	D1
chan7 ---->	third	8	D0

acknowledge delays and the delays between successive word, and byte transmissions, by making timing diagrams of 4 and 8 channels in parallel. This, however, is left as a suggestion for future research.

Conclusion 3

The Links really are able to operate in parallel!

4. **Comparison Between the Four Constructs**

TABLE 8

EFFECT OF PARALLELISM ON TRANSFER RATES FOR 10000 BYTES BLOCK SIZE **

	input/output primitives		BYTE SLICE procedure	WORD SLICE procedure
	bytes	words		
1 channel	370	1510	3670	3670
2 channels	190	770	3670	3670
3 channels	160	640	3650	3650
4 channels	160	640	3630	3620

** Values are in kbytes/sec rounded to tenths.

Table 8 shows the results obtained for 2, 3 and 4 links transmitting in parallel for each of the constructs.

These results where obtained using the Link Evaluation Program for all the constructs, listed on Appendix E, with no special priority for communications, and with no other processes being executed on the cpus, besides the transmitter and receiver processes. The time measurements were made at the transputer root at the B001 board. It is clear for us that although the BYTE.SLICE and WORD.SLICE procedures are not affected for more channels in parallel for this block size, the input and output primitives indeed are, but this will be addressed in the next section. It is still worthy of mention that several attempts were made to increase the transfer rate of the primitives input and output by using different loop sizes, no loops at all, different number of bytes, or words after each ? or ! separated by colons but in none of these cases a significant improvement was noticed.

D. MESSAGE SIZE AND CHANNEL PARALLELISM INFLUENCE.

Once we overcame the first phase of the research, validating the software we were using, we moved our attention towards the fourth and fifth research questions:

- What is the effect of message lengths on the link transfer rates?
- What is the mutual effect, on the link transfer rates, of more links operating in parallel.

To address these topics, The Link Evaluation Program was designed and implemented, using the programming concepts presented on previous sections of this chapter. What it does basically is, after the user's choice of type of construct and existence or non of concurrent process running on the CPU of the communicating transputers, named "cpumode", it builds a Table showing the transfer rates for the 16 different message sizes and 9 different channel parallelism cases, for the chosen option, and prompts the user for a new run. Appendix E presents the program, written in Occam, but one doesn't need to understand the program to grasp the results obtained, that will be presented in the following subsections, and in the next chapter. The configuration used for this program was the same of Figures 3.14 and 3.17 .

1. How to Read the Tables

The tables have ten (10) columns as follows:

- BYTES - Shows the number of bytes transmitted for the results obtained in that row.
- 1 OUT - Results obtained measuring transmission through only one channel from root to transputer 0.

- 1 IN - The same as above for reception on the root from transputer 0
- 2 OUT - Results obtained measuring transmission in parallel through two channels from root to transputer 0 and transputer 1.
- 2 IN - Same as above for reception in parallel.
- 3 OUT - Results obtained measuring transmission in parallel from root to transputers 0, 1 and 2.
- 3 IN - Same as above for reception in parallel.
- 4 OUT - Results obtained measuring parallel transmission from root to transputers 0, 1, 2 and 3.
- 4 IN - Same as above for reception in parallel.
- 4 IN/OUT - Results obtained measuring transmission and reception in parallel to/from transputers 0, 1, 2 and 3, using all 8 channels from the four links that exist in one transputer.

TABLE 9

TRANSPUTER LINK TRANSFER RATE
BYTE SLICE (1) - NO CONCURRENT PROCESS - 10 MBITS/SEC

BYTES	1 OUT	1 IN	2 OUT	2 IN	3 OUT	3 IN	4 OUT	4 IN	4INOUT
1	625	616	250	250	200	198	161	161	98
2	1217	1237	500	500	400	400	325	333	196
4	1531	2130	779	1000	648	788	650	646	384
8	2183	2811	1570	1582	1311	1301	1085	1096	690
16	2758	2924	2101	2222	1948	1919	1702	1694	1255
32	3224	3246	2589	2800	2482	2544	2330	2398	1835
64	3427	3646	3116	3226	2942	3048	2817	2954	2462
128	3543	3644	3332	3497	3265	3390	3187	3320	2945
256	3605	3741	3496	3656	3444	3596	3398	3558	3231
512	3635	3778	3578	3733	3555	3697	3509	3677	3401
1024	3650	3754	3627	3741	3604	3712	3575	3702	3512
1280	3654	3748	3640	3742	3611	3713	3587	3698	3529
2048	3658	3740	3652	3738	3621	3715	3604	3703	3549
4096	3662	3735	3663	3733	3634	3720	3618	3709	3573
8192	3665	3732	3668	3732	3645	3721	3627	3714	3585
10000	3667	3731	3669	3730	3647	3721	3623	3717	3591

* Values in kbits/sec

Some tables, though, have three different columns labeled:

- 1 IN/OUT - instead of 1 IN
- 2 IN/OUT - instead of 2 IN
- 3 IN/OUT - instead of 3 IN

In this columns, as the reader may have guessed already, the results presented refer to the transputer root transmitting and receiving at the same time through the number of links specified.

For each of the constructs the results are presented in table format and when necessary a graphical representation of the table. Each individual number on the tables is the average of 20 sequential runs. The results are in kbits/sec due to non availability of floating point for Proto-Occam and our need for precision.

2. **BYTE SLICE Procedure**

Table 9 and 10 show us the transfer rates obtained for this procedure with communication being the only process being executed by the transputers involved. Figure 3.22 is a graphical representation of Table 9 .

TABLE 10

TRANSPUTER LINK TRANSFER RATE
BYTE SLICE (2) - NO CONCURRENT PROCESS - 10 MBITS/SEC

BYTES	1 OUT	1INOUT	2 OUT	2INOUT	3 OUT	3INOUT	4 OUT	4 IN	4INOUT
1	625	248	250	166	194	125	161	166	98
2	1250	500	500	333	400	245	322	333	196
4	1518	793	779	651	645	487	645	648	384
8	2201	1585	1567	1099	1318	851	1101	1105	689
16	2782	2155	2208	1711	1951	1458	1701	1714	1256
32	3227	2636	2702	2326	2503	2133	2314	2379	1837
64	3513	3067	3116	2850	2988	2667	2813	2975	2741
128	3578	3350	3368	3204	3305	3075	3186	3320	2926
256	3628	3491	3522	3417	3476	3333	3404	3538	3230
512	3663	3574	3601	3532	3578	3483	3516	3662	3404
1024	3684	3617	3651	3587	3632	3568	3581	3683	3487
1280	3687	3624	3657	3591	3640	3584	3595	3685	3510
2048	3692	3636	3672	3597	3653	3603	3617	3689	3543
4096	3694	3648	3683	3616	3671	3623	3624	3692	3576
8192	3698	3655	3690	3619	3678	3632	3632	3694	3596
10000	3699	3657	3692	3614	3679	3638	3629	3696	3597

* Values in kbits/sec

From Table 9 we can notice the overall tendency of input be quicker than output, due to the way the timers are started and stopped by the flags. The flags used in this and all following tables were placed from the B003 transputers to the root. When we had the flags inverted the values had a tendency to be bigger for the output, so one may disregard the difference. For this reason we tried to show most of times the values for "in/out" instead for "in". Most important, however is the effect of message

size and channel parallelism reducing the transfer rates sensibly for smaller message sizes, but with insignificant effect for message sizes above 256 bytes.

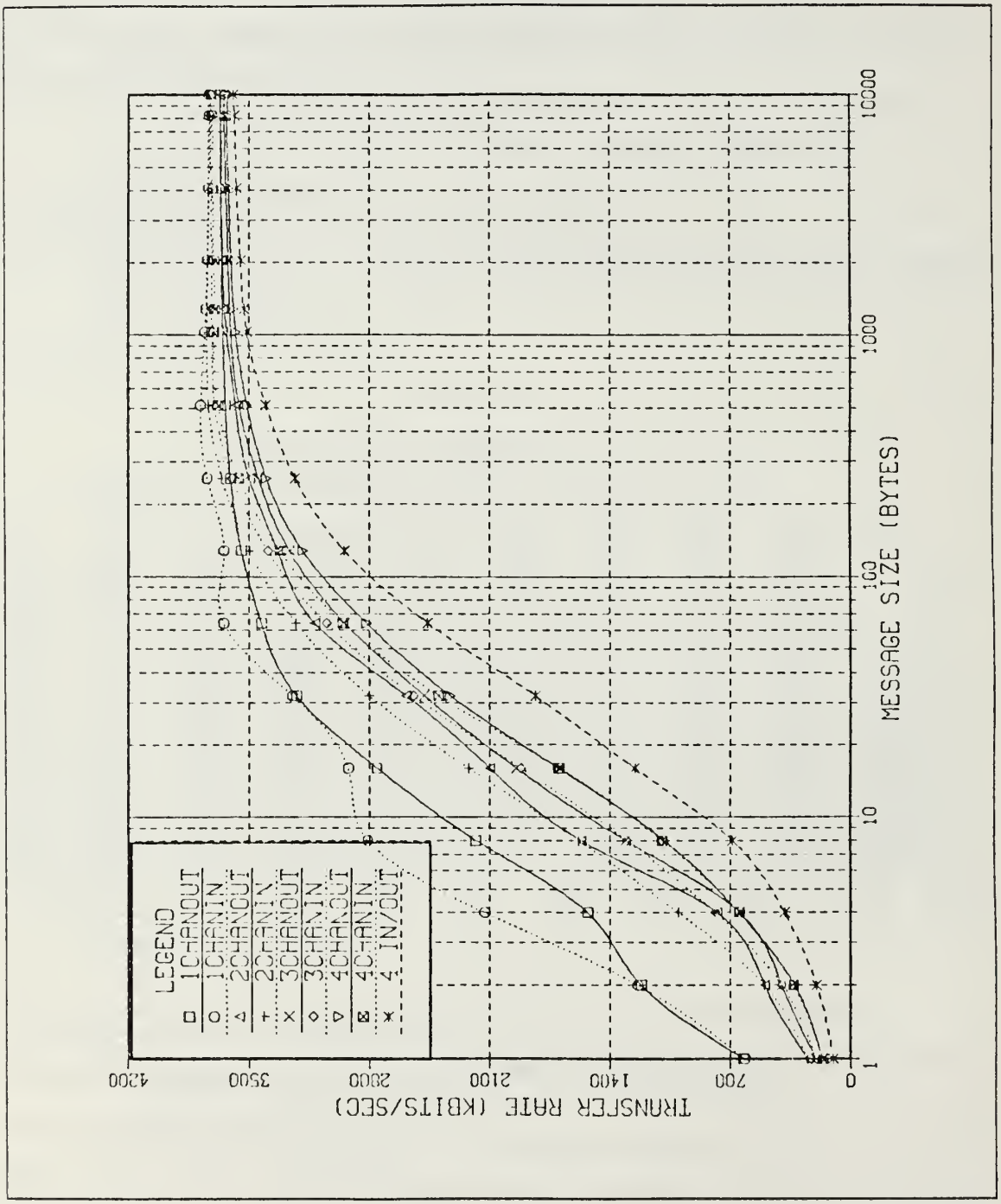


Figure 3.22 Transputer Link Transfer Rate
Byte Slice - No Process in Parallel - 10 mbits/sec.

3. WORD SLICE Procedure

TABLE 11

TRANSPUTER LINK TRANSFER RATE -
WORD SLICE - NO CONCURRENT PROCESS - 10 MBITS/SEC

BYTES	1 OUT	1 IN	2 OUT	2 IN	3 OUT	3 IN	4 OUT	4 IN	4INOUT
4	1287	1868	666	811	533	625	452	512	294
8	1910	2513	1333	1330	1063	1061	898	890	540
16	2580	3025	1985	1956	1667	1682	1454	1466	998
32	3083	3377	2541	2588	2284	2300	2077	2230	1632
64	3321	3559	2956	3091	2830	2875	2647	2790	2266
128	3491	3679	3294	3406	3156	3278	3068	3213	2783
256	3572	3738	3492	3598	3401	3521	3339	3485	3148
512	3617	3771	3571	3707	3530	3662	3479	3634	3373
1024	3644	3754	3624	3735	3589	3694	3557	3679	3486
1280	3648	3739	3638	3736	3594	3699	3570	3680	3516
2048	3655	3740	3648	3734	3613	3708	3595	3694	3546
4096	3662	3733	3659	3731	3638	3714	3609	3705	3566
8192	3664	3730	3669	3730	3642	3717	3621	3711	3586
10000	3666	3730	3669	3729	3645	3718	3622	3714	3588

* Values in kbits/sec

As seen in Table 11, the results obtained from WORD SLICE are very similar to the ones we had got for the BYTE SLICE procedure, so if the reader wants, he or she may use the same Figure 3.22 to have a better feeling of what these numbers means. All comments made for the BYTE SLICE procedure are valid also for WORD SLICE.

Conclusion 4

Message size has a major effect reducing the transfer rate for block transfers (BYTE SLICE and WORD SLICE).

4. Input and Output Primitives

a. Transmitting and Receiving Bytes

Table 12 shows us the results using the primitives input and output to transmit and receive bytes. As we can see, there is no variation as the number of bytes increase. This is due to the fact that each byte is transmitted individually as can be seen in Figure 3.12 . We can also notice that there is a significant decrease as more

TABLE 12

TRANSPUTER LINK TRANSFER RATE -
INPUT/OUTPUT (BYTES 1) - NO CONCURRENT PROCESS -
(10 MBITS/SEC)

BYTES	1 OUT	1 IN	2 OUT	2 IN	3 OUT	3 IN	4 OUT	4 IN	4INOUT
1	370	547	192	227	156	179	156	147	89
2	370	436	188	229	153	181	156	149	89
4	377	492	192	232	156	183	157	149	89
8	373	480	190	231	155	183	156	148	89
16	375	510	191	231	155	183	157	148	89
32	374	511	191	231	155	183	157	148	89
64	374	504	191	231	155	183	157	148	89
128	374	506	191	231	155	183	157	148	89
256	374	505	191	231	155	183	157	148	89
512	374	506	191	231	155	183	157	148	89
1024	374	506	191	231	155	183	157	148	89
1280	374	506	191	231	155	183	157	148	89
2048	374	505	191	231	155	183	157	148	89
4096	374	510	191	231	155	183	157	148	89
8192	374	510	191	231	155	183	157	148	89
10000	374	510	191	231	155	183	157	148	89

* Values in kbits/sec

channels are transmitting in parallel. We mention again that we tried several loop sizes or even no loop at all, with bytes separated by semicolons, but the results we have got where never significantly bigger than the ones presented. Table 13 stress the comparison when both channels of a same link are operating at the same time, transmitting and receiving messages. Note how the results on columns 3, 5, and 7 of Table 12 are 50% to 100% bigger than the ones from Table 13 .

b. Transmitting and Receiving Words (Integers)

Table 14 shows us the results for transmitting integers and we can notice again that message size does not affect the transfer rate, but more channels operating in parallel do. As we should expect from the previous results presented, this rate is, on the average, 4 times larger than the one for transmitting bytes.

Table 15 shows the comparison when both channels of a same link are transmitting and receiving at the same time. Again we confirm that, in terms of link performance, worse than having two different links transmitting at the same time, is to have the same link transmitting and receiving.

TABLE 13

TRANSPUTER LINK TRANSFER RATE -
INPUT/OUTPUT (BYTES/2) - NO CONCURRENT PROCESS -
(10 MBITS/SEC)

BYTES	1 OUT	1INOUT	2 OUT	2INOUT	3 OUT	3INOUT	4 OUT	4 IN	4INOUT
1	370	236	189	151	156	112	156	147	76
2	370	232	188	149	153	112	156	149	89
4	370	235	192	150	157	112	157	149	89
8	373	235	190	150	155	113	156	148	89
16	373	235	190	150	155	113	156	148	89
32	373	235	190	150	155	113	156	148	89
64	374	235	191	151	155	113	156	148	89
32	374	235	191	151	155	113	156	148	89
64	374	235	191	151	155	113	157	148	89
128	374	235	191	150	155	113	157	148	89
256	374	235	191	150	155	113	157	148	89
512	374	235	191	151	155	113	156	148	89
1024	374	235	191	151	155	113	157	148	89
1280	374	235	191	151	155	113	157	148	89
2048	374	235	191	151	155	113	156	148	89
4096	374	235	191	151	155	113	157	148	89
8192	374	235	191	150	155	113	157	148	89
10000	374	235	191	150	155	113	157	148	89

* Values in kbits/sec

Conclusion 5

More channels in parallel has a great reducing effect over the transfer rate for all constructs except block transfers (BYTE and WORD SLICE), bigger than 256 bytes.

This conclusion does not contradict Conclusion 3, but reduces the universe in which that is applicable.

TABLE 14

TRANSPUTER LINK TRANSFER RATE -
INPUT/OUTPUT (WORDS 1) - NO CONCURRENT PROCESS -
(10 MBITS/SEC)

BYTES	1 OUT	1 IN	2 OUT	2 IN	3 OUT	3 IN	4 OUT	4 IN	4INOUT
4	1526	2330	769	1000	643	785	628	640	377
8	1491	2369	763	1000	634	770	631	634	375
16	1484	2290	761	1000	635	769	629	640	375
32	1509	2326	765	1000	635	769	629	640	375
64	1504	2321	767	1003	640	772	635	642	376
128	1505	2367	766	1003	640	772	635	642	377
256	1509	2366	767	1004	641	772	635	642	376
512	1508	2383	767	1004	641	773	635	642	377
1024	1509	2382	767	1004	641	773	635	642	376
1280	1508	2384	767	1004	641	774	635	642	377
2048	1509	2384	767	1004	641	774	635	642	377
4096	1511	2396	767	1004	641	774	635	642	377
8192	1510	2394	767	1005	641	774	636	643	377
10000	1509	2394	767	1005	641	774	636	643	377

* Values in kbits/sec

TABLE 15

TRANSPUTER LINK TRANSFER RATE -
INPUT/OUTPUT (WORDS 2) - NO CONCURRENT PROCESS -
(10 MBITS/SEC)

BYTES	1 OUT	1INOUT	2 OUT	2INOUT	3 OUT	3INOUT	4 OUT	4 IN	4INOUT
4	1428	959	769	645	638	476	625	645	377
8	1481	959	769	634	634	470	621	634	375
16	1495	963	761	637	636	470	620	634	376
32	1509	969	765	640	640	471	622	637	376
64	1506	969	767	640	640	472	623	637	376
128	1505	969	767	640	640	472	623	637	376
256	1509	969	767	640	641	473	624	638	376
512	1509	970	767	641	641	473	624	638	377
1024	1509	970	767	641	641	473	624	638	377
1280	1509	970	767	641	641	473	624	637	377
2048	1510	970	767	641	641	473	624	638	377
4096	1510	971	767	641	641	473	624	637	377
8192	1510	971	767	641	641	473	624	638	377
10000	1510	971	767	641	641	473	624	637	377

* Values in kbits/sec

IV. THE MUTUAL EFFECTS BETWEEN PROCESSOR AND THE FOUR LINKS

When using the transputer in a multi-transputer configuration, most likely it will be necessary in each transputer node, at least one process to route messages between transputers, and another to execute some processing task. Our role in this chapter is to examine how a process task oriented would affect a routing process, changing the transfer rate on the links. Also, we are going to analyze how a routing process handling large messages may affect the throughput of a computation bound process.

A. EFFECT OF CONCURRENT PROCESSES OVER COMMUNICATIONS

1. Initial Considerations

This section addresses the sixth and seventh research questions as follows:

- Can the CPU execute a process in parallel with some or all the links operating?
- What is the effect of a communication independent process running on the CPU, over the transfer rates obtained in a link by another process in this transputer?

To observe this effect with the links working at 10 mbits/sec rate, the same Evaluation Program was used, but using different program defined cpu modes. Figure 4.1 shows the CPU modes made available by the program to the user's choice.

```
0 - No concurrent process in the cpus
1 - B003 cpus with sum process concurrently (par)
2 - all cpus with sum process concurrently (par)
3 - B003 cpus with sum process concurrently (pripar)
4 - all cpus with sum process concurrently (pripar)
5 - B003 cpus with array product process concurrently (par)
6 - all cpus with array product process concurrently (par)
7 - B003 cpus with array product process concurrently(pripar)
8 - all cpus with array product process concurrently (pripar)
```

Figure 4.1 CPU modes Available in the Link Evaluation Program.

Two procedures called "cpubusysum" or "cpubusyprod" would be running concurrently with the transmitter and receiver procedures in one or both communicating CPUs according to the CPU mode chosen and with the following effects:

- "cpubusysum" - This procedure would initiate at the start of communications and execute sum operations continuously, until the communications were finished, with few memory accesses involved.
- "cpubusyprod" - This procedure, equally, would initiate at the start of communications and execute array products continuously until communications were finished. Now 100 times more memory accesses was necessary.

Figure 4.2 shows the code to permit this (e. g. transmission), for a WORD SLICE construct. Similar code exists for the other constructs, only changing the procedure "wordtransfer" to the applicable one. See Appendix E for more details on them.

```
SEQ   --- main word.slice.transfer
  -- word buffers  initialization
  SEQ k = [1 FOR maxwordblock.size]
    SEQ
      wbuffer0 [k] := 10000
      wbuffer1 [k] := 20000
      wbuffer2 [k] := 30000
      wbuffer3 [k] := 40000
  SKIP
  IF
    cpumode = '2'
      PAR
        wordtransfer (repetition, cpumode, flag, counter)
        cpubusysum (flag, counter)
    cpumode = '4'
      PRI PAR
        wordtransfer (repetition, cpumode, flag, counter)
        cpubusysum (flag, counter)
    cpumode = '6'
      PAR
        wordtransfer (repetition, cpumode, flag, counter)
        cpubusyprod (flag, counter)
    cpumode = '8'
      PRI PAR
        wordtransfer (repetition, cpumode, flag, counter)
        cpubusyprod (flag, counter)
    TRUE
      wordtransfer (repetition, cpumode, flag, counter):
```

Figure 4.2 How the Concurrent Processes Were Called.

2. **Process Priority Considerations**

The transputer supports two priority levels built in in hardware:

1 Priority 0 (High) - processes with this priority are executed always, without being interrupted until they finish. They should work only for a short period of time because if the sum of time spent by all priority processes is greater than a time slice, the low priority processes will not be able to proceed [Ref. 7: section 2, p.3]. The high priority processes preempt the low priority ones.

2 Priority 1 (Low) - These are executed when no more high priority processes are able to proceed, in a time slice fashion of 1 msec for each process.

In our program, considering the two processes to be executed in the same CPU, three situations were examined. Assuming the processes names are for example "communication" and "cpubusy" we have:

a. both processes under a PAR construct - in this case processes will be time sliced, because both are low priority, at every 1 msec. This case was observed by using "cpumode" 1, 2, 5 and 6 in the Link Evaluation Program.

b. both processes under a PRI PAR with communications in high priority - in this case communication will always be executed at once. Remember that it took 31.5 msec for a 15,000 bytes block to be transmitted, and the time slice is 1 msec, and so the cpubusy will not have a chance to be executed if communication is going on, unless the number of bytes transmitted is smaller than 475 :

   - (475 x 8) / 3,800,000 = 0.001 sec or 1 msec, if we considered a rate of 3.8 mbits/sec. This cases were observed by using "cpumode" 3,4,7 and 8 in the Link Evaluation Program.

c. both processes under PRI PAR but with the cpubusy process in high - in this case the communications never occurred because the "cpubusy" process although started together with the communications, should be stopped by a flag of that process (by design), that could never come, because the process was never being granted CPU time. This is why no mention to this case is made on the Link Evaluation Program.

Again, analyzes were made for the four constructs and the results are presented in tables and graphics similar to the ones in the previous chapter.

Another point to mention is that when placing a concurrent process in only one of the communicating CPUs, the B003 transputer was the chosen one, because of its higher internal clock. When the B001 transputer was with the "cpubusy" process first, no changes were noticed in the transfer rates as we added a cpubusy process on the B003 transputers. In the way we did, we could clearly see the two step change.

3. **BYTE SLICE Procedure**

a. Using the PAR Construct

(1) *One Transputer Only (cpumode = 1 or 5).*

In this case Table 16 for "cpumode = 1" and Table 17 for "cpumode = 5", shows us the results, and Figure 4.3 is the graphical representation of Table 16 .

We can observe that when the CPU has a concurrent process running with the same priority as the communications process, the transfer rate is reduced from 10% to 99.5% less of the original values on Table 9 .

TABLE 16

TRANSPUTER LINK TRANSFER RATE - BYTE SLICE - PROCEDURE CPUBUSYSUM CONCURRENT AT THE B003 - 10MBITS/SEC

BYTES	1 OUT	1INOUT	2 OUT	2INOUT	3 OUT	3INOUT	4 OUT	4 IN	4INOUT
1	3	1	3	1	3	1	3	3	1
2	7	3	7	3	7	3	7	7	3
4	15	7	15	7	15	7	15	15	7
8	31	15	31	15	31	15	31	30	15
16	61	30	61	30	61	30	61	61	30
32	128	61	121	61	120	61	120	120	61
64	253	121	235	120	234	120	234	234	121
128	597	234	441	234	442	234	442	445	234
256	789	441	792	441	804	441	801	811	440
512	1311	818	1326	788	1348	788	1337	1320	786
1024	2010	1315	1954	1317	1969	1315	1957	1994	1313
1280	2204	1518	2181	1494	2123	1489	2121	2142	1489
2048	2546	1938	2561	1916	2552	1934	2550	2588	1929
4096	3013	2535	3030	2535	2999	2527	2979	3017	2517
8192	3324	3004	3316	2976	3312	2967	3280	3326	2955
10000	3386	3100	3380	3083	3370	3065	3332	3385	3051

* Values in kbits/sec.

TABLE 17

TRANSPUTER LINK TRANSFER RATE - BYTE SLICE - PROCEDURE CPUBUSYPROD CONCURRENT AT THE B003 - 10MBITS/SEC

BYTES	1 OUT	1INOUT	2 OUT	2INOUT	3 OUT	3INOUT	4 OUT	4 IN	4INOUT
1	3	1	3	1	3	1	3	3	1
2	7	3	7	3	7	3	7	7	3
4	15	7	15	7	15	7	15	15	7
8	31	15	31	15	31	15	31	30	15
16	61	30	61	30	61	30	61	61	30
32	128	61	121	61	120	61	120	121	61
64	253	121	235	120	234	120	234	235	121
128	479	234	443	234	443	234	445	444	234
256	854	441	805	441	804	441	804	810	441
512	1402	818	1349	790	1338	789	1339	1327	788
1024	1975	1316	1973	1318	1971	1318	1924	1971	1295
1280	2151	1492	2132	1496	2123	1490	2110	2144	1490
2048	2568	1938	2573	1938	2526	1929	2515	2544	1923
4096	3036	2539	3006	2529	2999	2510	2982	3020	2498
8192	3332	2989	3320	2989	3311	2965	3281	3355	2964
10000	3396	3091	3382	3076	3374	3068	3343	3417	3060

* Values in kbits/sec.

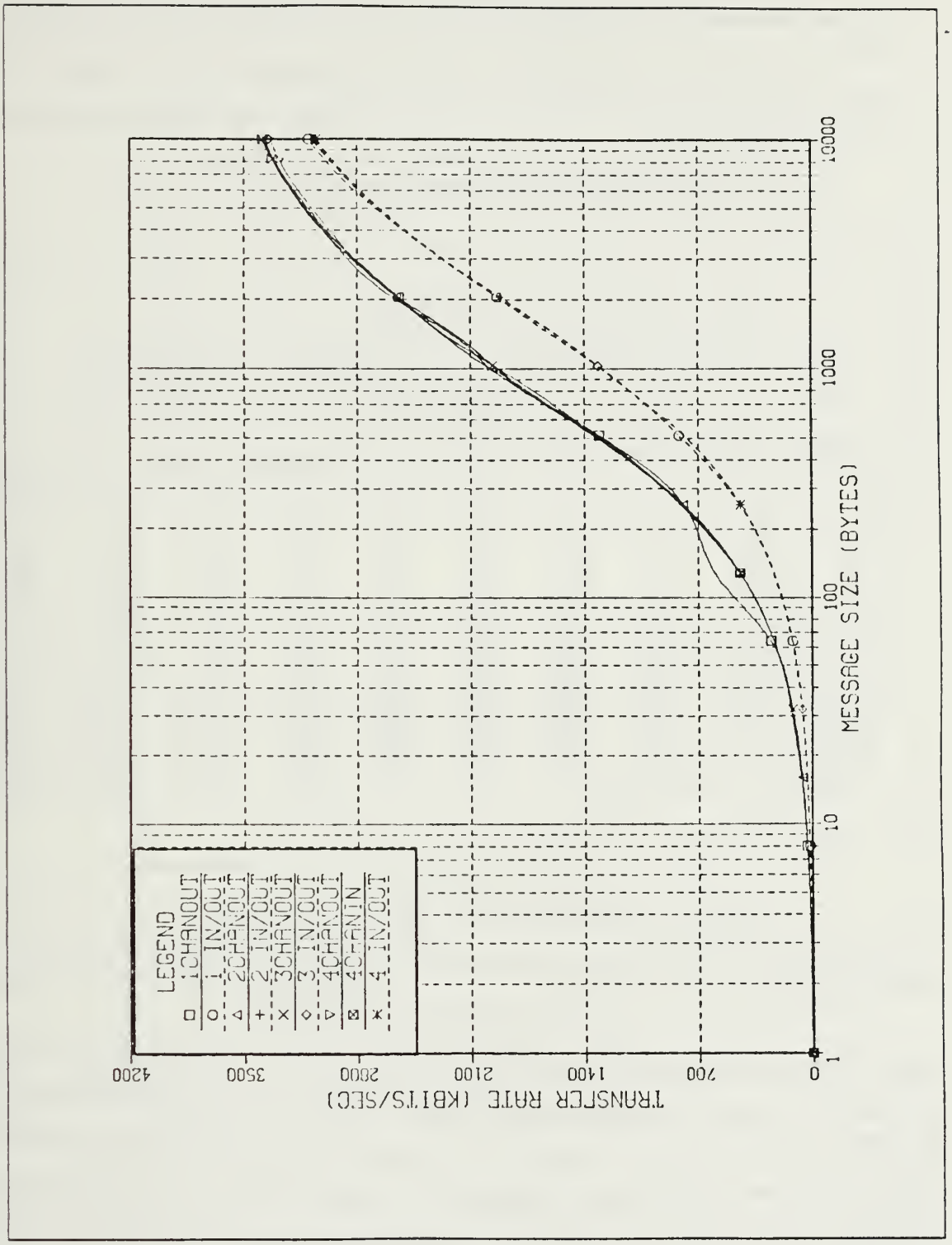


Figure 4.3 Transputer Link Transfer Rate - BYTE SLICE
Procedure Cpubusysum Concurrent at the B003 - 10 mbits/sec.

This is a great surprise for us because we are only timing the communication itself and although we can not prove, it looks like the communication process is alive and sharing CPU time with the cpubusy process, instead of being inactive while the links communicate, as all the references led us and our predecessors to believe [Ref. 5: p. 16].

(2) *Both Transputers Busy (cpumode = 2 or 6).*

TABLE 18

TRANSPUTER LINK TRANSFER RATE - BYTE SLICE - PROCEDURE CPUBUSYSUM CONCURRENT AT ALL CPUS - 10MBITS/SEC

BYTES	1 OUT	1INOUT	2 OUT	2INOUT	3 OUT	3INOUT	4 OUT	4 IN	4INOUT
1	2	1	1	1	1	1	1	1	1
2	4	2	2	2	2	2	2	2	2
4	9	4	4	4	4	4	4	4	4
8	19	9	9	9	9	9	9	9	9
16	39	19	19	19	19	19	19	19	19
32	78	39	39	39	39	39	39	39	38
64	156	78	78	78	78	77	77	78	77
128	312	156	156	156	156	156	156	156	155
256	624	312	312	312	312	312	312	312	311
512	1249	624	624	624	624	624	624	624	623
1024	2498	1248	1249	1248	1248	1248	1249	1249	1247
1280	3120	1560	1560	1561	1561	1561	1561	1561	1559
2048	2498	1665	1665	1665	1666	1666	1665	1665	1664
4096	3332	2498	2499	2499	2499	2499	2499	2499	2497
8192	3332	2855	2856	2856	2856	2856	2856	2856	2855
10000	3487	3050	3050	3050	3050	3050	3050	3050	3049

* Values in kbits/sec.

Table 18 and Figure 4.4 need no explanation. The results for "cpubusyprod" are not presented because they happen to give us exactly the same results for "cpubusysum", as we saw in the previous subsection.

One may notice in Table 18 column "1 OUT", that the value for 2048 bytes (2498) is a lot smaller than the previous one (3120), and the effect is clearly seen in Figure 4.4 . What may be happening is that as the buffer declared on the program (buffer0) may have the initial bytes of it in the internal memory of the transputers (2 kbytes), and when external memory begins to be accessed, the transfer rate goes down, or reduce the rate of increase, as we can see on the lower curve of figure 4.4, where all the remaining curves coincide and have a brake on the rate of increase at the same point. This is what the author thinks is happening but we were not able to prove it.

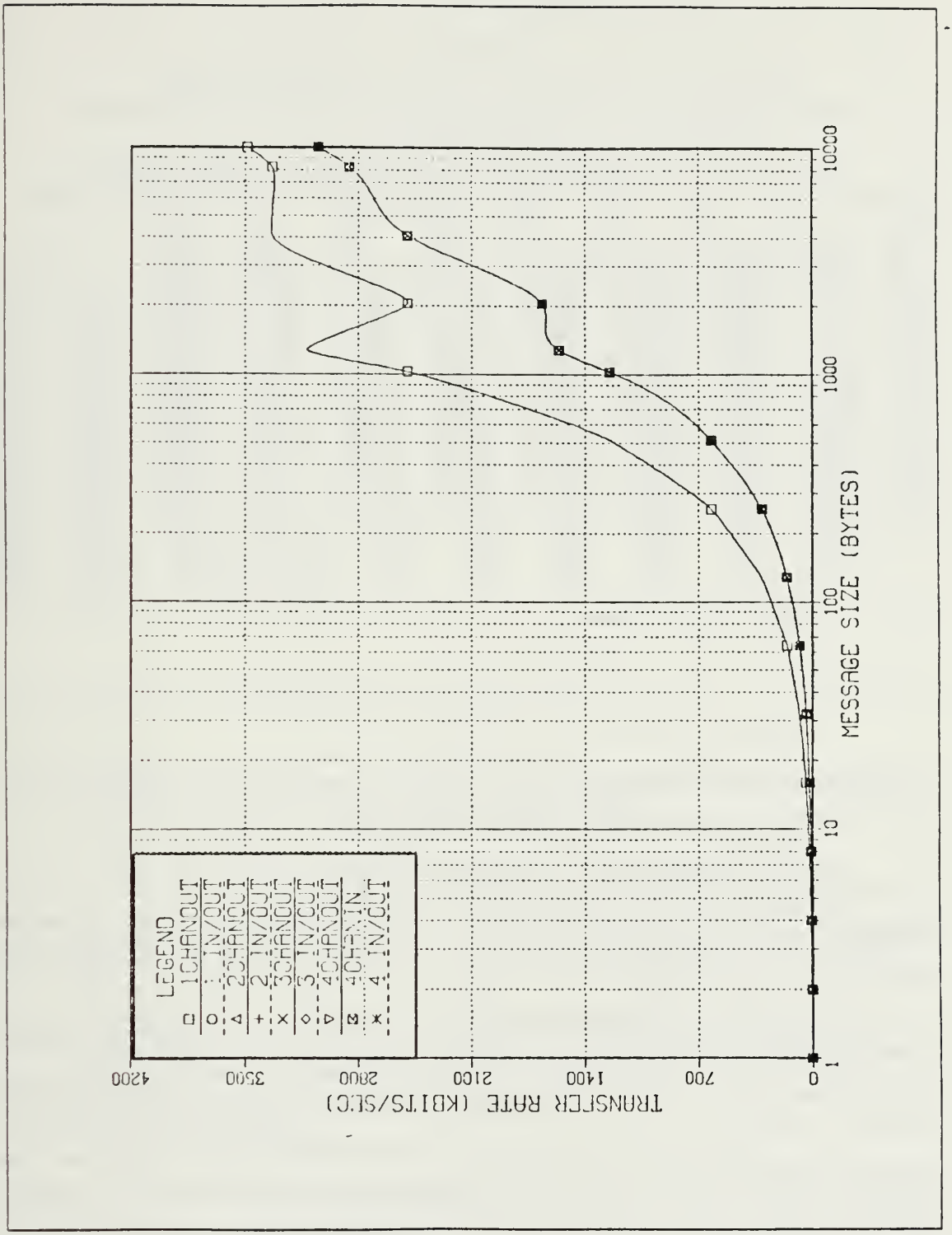


Figure 4.4 Transputer Link Transfer Rate - BYTE SLICE
Procedure Cpubusysum Concurrent at All CPUs - 10 mbits/sec.

b. Using the PRI PAR Construct

TABLE 19

TRANSPUTER LINK TRANSFER RATE - BYTE SLICE - PROCEDURE CPUBUSYSUM CONCURRENT AT THE B003 (HIGH) - 10 MBITS/SEC

BYTES	1 OUT	1INOUT	2 OUT	2INOUT	3 OUT	3INOUT	4 OUT	4 IN	4INOUT
1	625	248	250	166	194	125	161	166	98
2	1250	500	500	333	400	245	322	333	196
4	1518	793	779	651	645	487	645	648	384
8	2201	1585	1567	1099	1318	851	1101	1105	689
16	2782	2155	2208	1711	1951	1458	1701	1714	1256
32	3227	2636	2702	2326	2503	2133	2314	2379	1837
64	3513	3067	3116	2850	2988	2667	2813	2975	2471
128	3578	3350	3368	3204	3305	3075	3186	3320	2926
256	3628	3491	3522	3417	3476	3333	3404	3538	3230
512	3663	3574	3601	3532	3578	3483	3516	3662	3404
1024	3684	3617	3651	3587	3632	3568	3581	3683	3487
1280	3687	3624	3657	3591	3640	3584	3595	3685	3510
2048	3692	3636	3672	3597	3653	3603	3617	3689	3543
4096	3694	3648	3683	3616	3671	3623	3624	3692	3576
8192	3698	3655	3690	3619	3678	3632	3632	3694	3596
10000	3699	3657	3692	3614	3679	3638	3629	3696	3597

* Values in kbits/sec.

Table 19 and and Figure 4.5 show the results for one concurrent process running in the B003 transputers, and Table 20 and Figure 4.6 the same for all CPUs with concurrent process but in all cases communication having the high priority.

As we see the figures are even better, on the average, than when no process was running concurrently, as seen on Table 9 . **This is why we believe and suggest that processes that handle only communications, as the routers, should be given always high priority.**

For each of the possible cases, Table 21 shows us the number of processes executed in parallel in each transputer. Although they do not have a valuable absolute meaning, they give us a comparative value of the behavior of the CPU in the different constructs. The reason for that is in the way the program was made. There are some intervals between the several communication sessions and repetitions, were the cpubusy process would be able to operate, time sliced with the calculations and output to screen, done after each of these sessions.

TABLE 20

TRANSPUTER LINK TRANSFER RATE - BYTE SLICE
PROCEDURE CPUBUSYSUM CONCURRENT AT ALL CPUS (HIGH) -
10 MBITS/SEC

BYTES	1 OUT	1INOUT	2 OUT	2INOUT	3 OUT	3INOUT	4 OUT	4 IN	4INOUT
1	1132	555	500	263	359	172	263	263	127
2	1728	981	948	526	690	353	520	526	256
4	2288	1540	1436	1036	1204	681	977	992	498
8	2684	2165	2115	1621	1831	1341	1565	1723	986
16	3203	2740	2649	2245	2435	1933	2199	2352	1664
32	3459	3136	3102	2792	2939	2526	2730	2912	2265
64	3549	3376	3375	3156	3276	2996	3139	3295	2788
128	3615	3511	3533	3399	3468	3303	3384	3517	3150
256	3655	3579	3600	3525	3568	3475	3516	3651	3359
512	3676	3621	3631	3589	3623	3565	3577	3720	3491
1024	3683	3641	3665	3623	3644	3619	3613	3714	3542
1280	3682	3646	3663	3630	3651	3626	3623	3707	3558
2048	3683	3651	3675	3641	3659	3641	3634	3702	3579
4096	3690	3657	3675	3650	3668	3649	3642	3701	3595
8192	3689	3661	3675	3655	3673	3653	3646	3698	3610
10000	3689	3663	3679	3655	3673	3654	3647	3698	3614

* Values in kbits/sec.

TABLE 21

NUMBER OF OPERATIONS EXECUTED CONCURRENTLY IN EACH
CPU*- BYTE SLICE USED

	Transputer b003		Transputer b001	
	cpu. sum	cpu. prod	cpu. sum	cpu. prod
1 PAR	2.7	6.5	inactive	inactive
2 PAR	5.0	11.9	3.9	7.8
1 PRI PAR	1.3	3.1	inactive	inactive
2 PRI PAR	1.3	3.1	0.9	1.8

* Values are in millions.

4. **WORD SLICE Procedure**

For the WORD SLICE Procedure, it happens that the results are very similar to the ones obtained for the BYTE SLICE Procedure and they will not be repeated

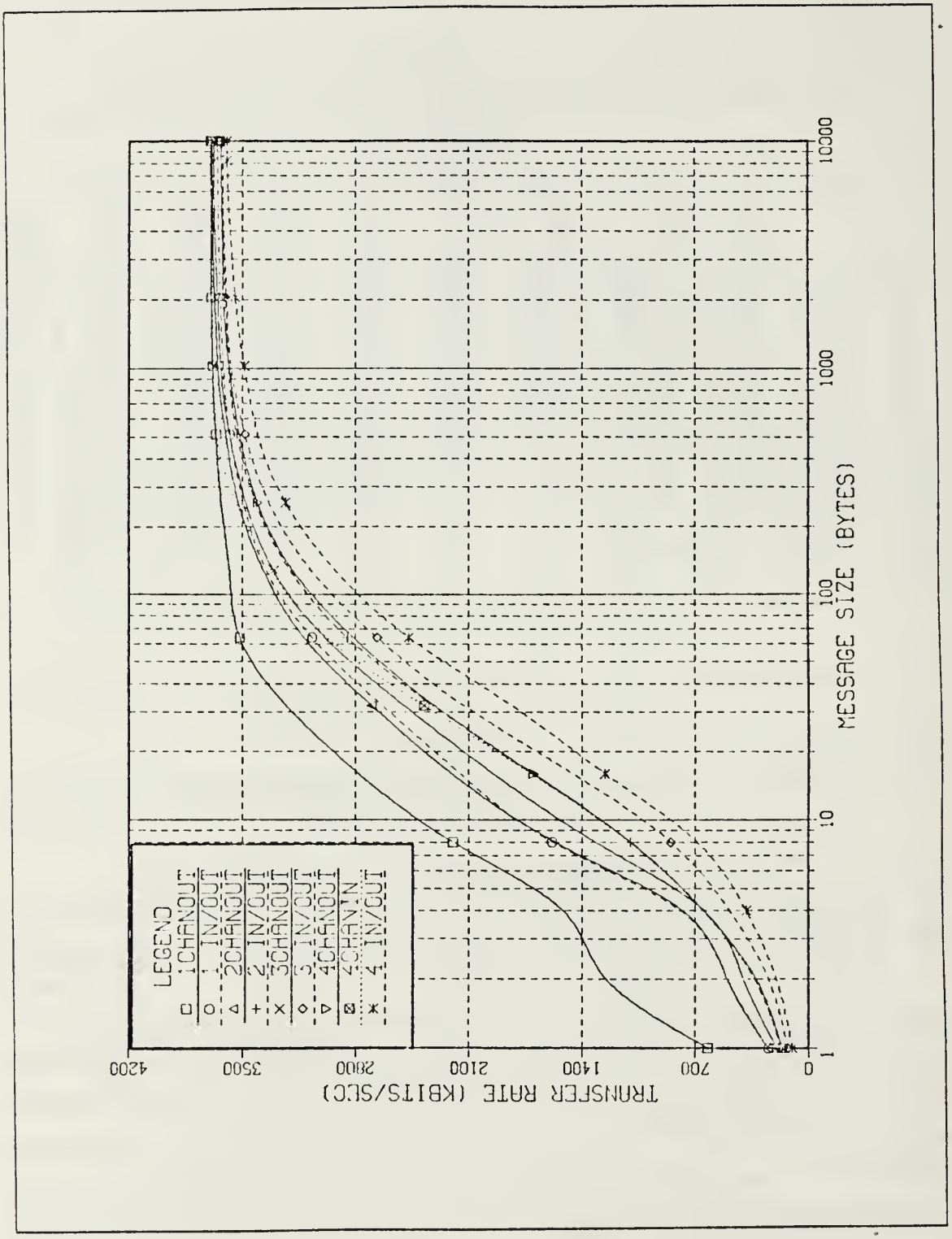


Figure 4.5 Transputer Link Transfer Rate - BYTE SLICE
Procedure Cpubusysum Concurrent at the B003(high) - 10 mbits/sec.

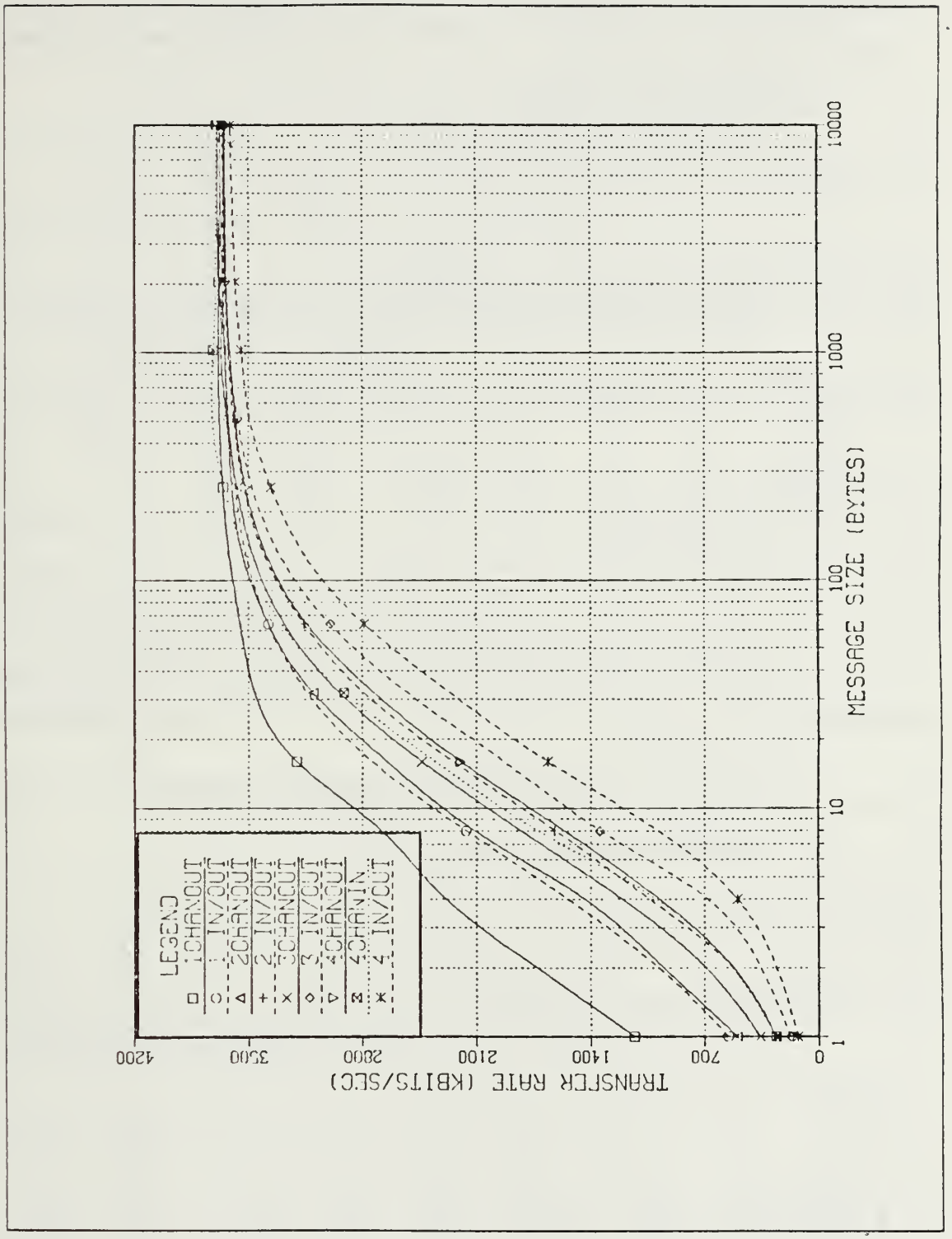


Figure 4.6 Transputer Link Transfer Rate - BYTE SLICE
Procedure Cpubusysum Concurrent at All CPUs(high) - 10 mbits/sec.

here. The reader may refer to all Tables and Figures described in the last section, just remembering that for word transfer the minimum number of bytes is 4, and so, the two first rows might be disregarded.

5. **Input and Output Primitives**

a. Transmitting and Receiving Bytes

TABLE 22

TRANSPUTER LINK TRANSFER RATE* - INPUT/OUTPUT (BYTES)
PROC CPUBUSYSUM CONCURRENT - 10 MBITS/SEC

cpumode	1 out	1in/out	2 out	2in/out	3 out	3in/out	4 out	4in/out
1 PAR	3	1	3	1	3	1	3	1
2 PAR	2	1	1	1	1	1	1	1
1 PRIPAR	370	230	190	150	160	110	155	90
2 PRIPAR	575	350	370	225	295	155	235	115

* Values in kbits/sec

Following a tendency observed before, there were no variations for transfer rates with respect to the message size. Table 22 shows us the figures obtained for the various priority schemes used.

These results were the same for the procedure "cpubusyprod", and for this reason are not shown.

b. Transmitting and Receiving Integers

TABLE 23

TRANSPUTER LINK TRANSFER RATE* - INPUT/OUTPUT (WORDS)
PROC CPUBUSY.SUM CONCURRENT - 10 MBITS/SEC

cpumode	1 out	1in/out	2 out	2in/out	3 out	3in/out	4 out	4in/out
1 PAR	15	7	15	7	15	7	15	7
2 PAR	9	4	4	4	4	4	4	4
1 PRIPAR	1510	970	765	640	640	470	625	375
2 PRIPAR	2345	1560	1450	1040	1200	650	930	480

* Values in kbits/sec

Table 23 shows us the results for transmitting and receiving integers with **input** and **output** primitives.

Several conclusions may be drawn from the two tables mentioned above:

- results for integers are in general four times larger than for bytes.
- A process running concurrently does affect the communications if under a PAR construct. Results are 50 to 100 times smaller than the ones obtained for no concurrent process using the CPU. See Table 13 .
- When running communications under PRI PAR on the B003 transputers, same results are obtained as with no other concurrent process. One shall compare third row of Table 22 (1 in/out), with Table 13 for bytes and third row of Table 23 with Table 15 for integers.
- When running communications in PRI PAR in both transputers the best transfer rates are obtained either for bytes or integers. So the concurrent CPU process will not affect the communications.

It is always good to remember that the cpu load cases examined are extreme cases that rarely or never will occur in any application program, but the results obtained, undoubtedly, show us a relation between cpu load and performance obtained on the links. So, referring back to research question 6, we are not able to affirm now if the links can operate in parallel with the processor, but next section will address this point again.

Conclusion 6

Under a PAR construct, a process working concurrently on the CPU, will reduce the transfer rate on the links.

Under the PRI PAR, it looks like the communication process in high priority does not suffer any dragging, but we have still a doubt of how much can a process do when the communications are in PRI PAR and are lengthy. This will be addressed in the next section.

B. THE EFFECT OF THE COMMUNICATIONS OVER CONCURRENT PROCESSES

This section addresses the eighth research question below:

- "What is the effect of the communications on the links, over a process that is being executed concurrently on the main processor of the same transputer?"

1. Initial Considerations

To observe this we needed to time a fixed length process without any communications occurring in the processor in which it was being executed, and time it later with communications in parallel through the links. As we mentioned before, in the latter case we needed to make sure that only the communications were happening concurrently, hopefully in parallel, in order to guarantee that the process being timed was not being dragged by other processes besides communication processes.

```
PROC counter (CHAN in,out, VALUE tnumber) =
-- description
---*********************************************************
--- Sums up the first 100000 integers and add the transputer
--- number to the total
---*********************************************************
  DEF maxope = 100000:   --- number of operations done
  VAR ch,total :
  VAR startime3, endtime3:
  SEQ
    total := tnumber
    in ? ch
    TIME ? startime3
    SEQ i = [0 FOR maxope]
      total := total + i
    TIME ? endtime3
    out ! total;startime3;endtime3:
```

Figure 4.7 Procedure Counter.

What was done, then, was to make a simple procedure called "counter" listed on Figure 4.7, and place it in a transputer with no other process. For this purpose, a transputer in a B003 board would be more appropriate, because we now are going to time the processor itself and performance could be affected in the B001 board by the terminal_driver, user interface and so forth. It is never repetitive to remember that with the links measurements these effects were not so strong because the links have constant speed of transmission, the 10 mbits/sec bit rate, independent of the processor internal cycle and load.

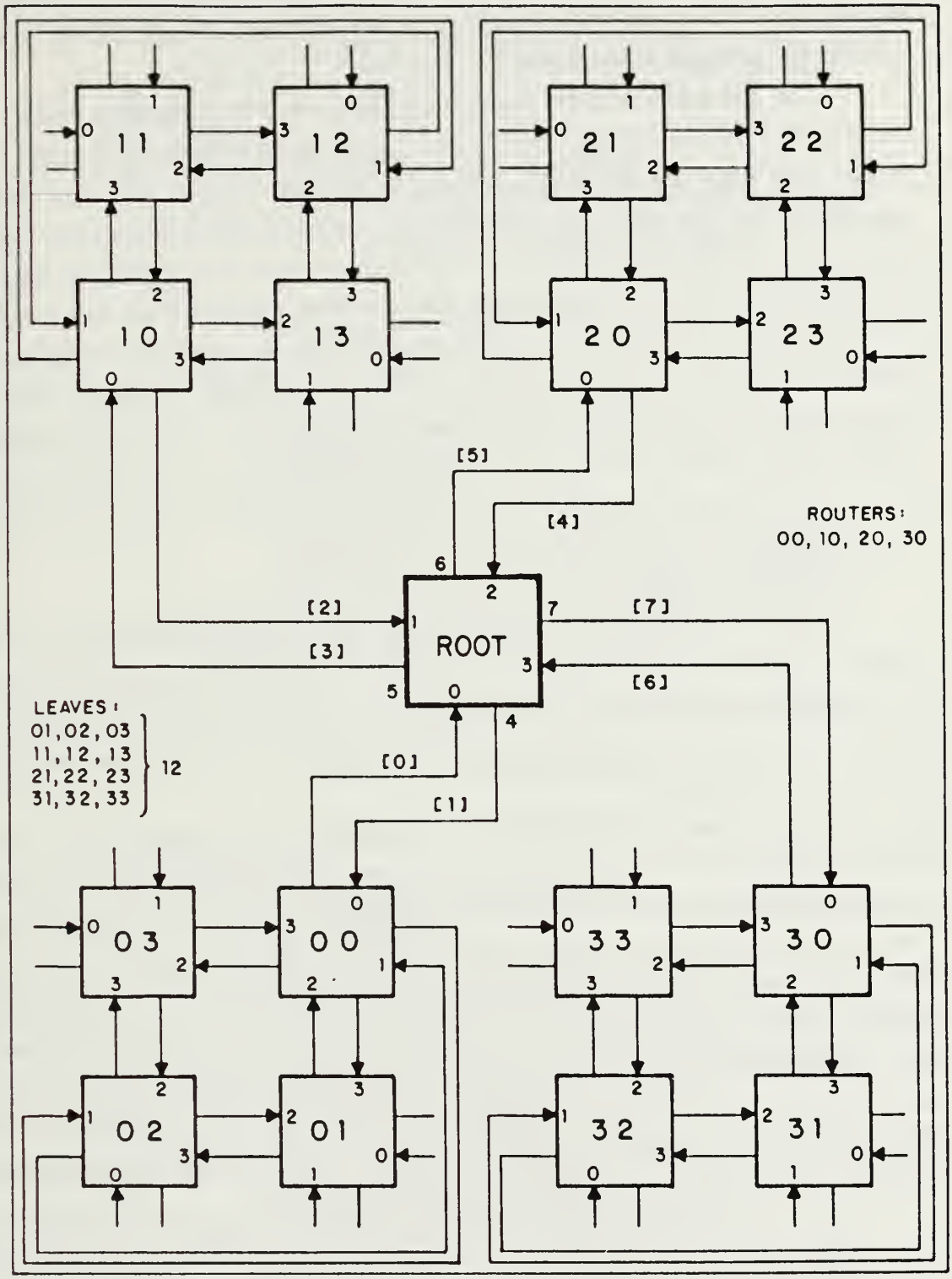


Figure 4.8 Configuration for Program Test Linearity (17).

The procedure "counter" sums up the first 100,000 integers plus the transputer number were it is located after receiving a flag, and send the result, startime and endtime out through a channel "out".

The harness where we placed this procedure was a program called "Test Linearity" that will be described now briefly. This program is listed in Appendix F and includes the main procedures, Host Proc, Route, and Counter, that are separate compilation (SC) and are placed in different transputers. The configuration used for this program is shown on Figure 4.8 . The procedure Host Proc is the user, keyboard and screen interfaces, and it is placed in transputer root. The procedure Route is placed in transputers 00, 10, 20 and 30, and executes in parallel the routing procedure and the counter. The remaning transputers (12) are all executing only the counter procedure. If we look close to the topology of the processors on Figure 4.8, we can see that we have a tree structure were the leaves are only executing counters, the second level nodes are the routers and the root is the host procedure (hostproc). Figure 4.9 lists Procedure route.

2. **Results Obtained**

The first measurement done, was the time to execute "counter" and we obtained 130 msec., using the tick.to.time routine to convert the tick values. This value was obtained in all 16 counters either alone in a transputer leaf or inside the routers, meaning that the present level of communications were not affecting the concurrent process on the routers CPU!

Then, to enforce a continuous communication, we placed in each router, besides the flags, and in parallel with the counter, three block transfers to the three leaves of each router using the BYTE.SLICE procedure with blocks of 50,000 first, and then varying from 70,000 to 256 bytes. If we recall from Chapter 3 this would assure us at least 105 and 147 msec, respectively for 50,000 and 70,000 bytes, of continuous communication, considering the rate of 3.8 mbits/sec. In fact, we also measured in this new configuration the time to execute the communication process alone, and it took respectively 103 and 144 msec, so implying a transfer rate of 3.88 mbits/sec for the three channels transmitting in parallel. This result a little bit higher than the ones observed in the last chapter is explained for two reasons:

1 The use of 15 Mhz transputers with shorter processes inside and so permitting most use of internal memory.

2 The bigger external memory permitting use of bigger block transfers.

The transfers were then timed in two modes:

```
PROC route(CHAN messagein,messageout,routeto1,routeto2,routeto3,
                  echofrom1,echofrom2,echofrom3,VALUE k)=
  DEF i = 4 :     --- number of counter procedures
  VAR msg :       --- flag
  VAR results[i] :
  VAR startime[i],endtime[i]:  --- timers
  CHAN softin,softout:  --- soft channels declared for
                        --- communication with procedure counter.
  -- SC PROC counter
  --- This procedure counter is listed in Figure 4.7
  SEQ
    PAR
      counter (softout,softin,k)
      -- routing process itself
      SEQ
        messagein ? msg
        SEQ
          PAR
            routeto1 ! msg
            routeto2 ! msg
            routeto3 ! msg
          softout ! msg
          PAR
            echofrom1 ? results-0-;startime-0-;endtime-0-
            echofrom2 ? results-1-;startime-1-;endtime-1-
            echofrom3 ? results-2-;startime-2-;endtime-2-
            softin ? results-3-; startime-3-;endtime-3-
    -- sending to the root results and timing
    SEQ i = [0 FOR 4]
      messageout ! results[i];startime[i];endtime[i]:
```

Figure 4.9 Procedure Route.

- 3 chanout, with three simultaneous BYTE.SLICE transmissions to the counters in different transputers,
- 3 in/out, with six simultaneous transfers (3 input and 3 output) to/from the counters in different transputers.

Note from the procedure route code in Figure 4.9 that a flag was sent to each transputer to make sure they were ready for the BYTE.SLICE transfer, and then another flag was sent to the local counter procedure and so do the best possible for the communications begin together with the local counter procedure.

As we can see from Table 26, for message blocks up to 520 bytes, no effect was noticed on the procedure counter! At this point no further investigation has occurred and two speculations could account for the observed data:

1 May be after 520 bytes long, the arrays being transmitted, begin to access external memory of the transputers. If this was the reason, the increase of time should be more proportional than the abrupt increase of 35% more in time (46/130) with an increase of 4.6% in the number of bytes transmitted (24/520) as shown in Table 25 .

TABLE 26

TIMMING OF PROCEDURE COUNTER

a. Message size 50,000 bytes

- time to execute communications only : 103 msec
- time to execute procedure counter with:
  1. No communications ..... 130 msec
  2. With 3 chanout..... 186 msec
  3. With 3 in/out ..... 195 msec

b. Variable message size with 3 chanout

	Timing of procedure counter(msec) in the router		in the leaves
bytes	par	pripar	both cases
70000	190	190	191
50000	186	186	191
10000	178	178	175
1000	176	176	130
544	176	176	130
528	156	156	130
520	130	130	130
< 256	130	130	130

2 The counter is being timed out, when communication takes more than 1 msec to finish - this looks more reasonable in the sense that if the time slice instead of 1 msec, that corresponds to 485 bytes to be transmitted at a 3.88 mbits/sec rate, is 1.07msec this would give us a transfer of 520 bytes in the period of a time slice because :

- (520 x 8) / 3,880,000 = 0.00107 sec or 1.07 msec

On the other hand, if we compare the total execution time of 195 msec for the worst case observed (process being executed concurrently), with the sum of the individual times necessary for counter or communications to accomplish its task , 233 msec, (See in Table 25 a.), we see a mismatch of 38 msec, when the two processes might be overlapping in time.

The great surprise, although was the unpredictable effect on the transputer leaves where we have the counter process executed sequentially after the communications and the timing only begins after the communications are over (Table 25 b, last column). We have no reasonable explanation for that.

So, as we see, no definitive conclusion of how the scheduling of the routing process and the counter process is happening, but from the times obtained, there has to exist some overlapping, but not total, between the counter and the routing processes in the router transputers. The results were consistent on the four routers.

Conclusion 7

The communication indeed affects the process being executed in the CPU, for messages greater than a threshold size. For our example this value was 520 bytes or bigger. Bellow this message size, communications had no effect over the process being executed on the CPU.

This first conclusion sure lead us to do a complete case study, on the subject matter varying the counter size, the message size and using another typical process instead of the simple counter, and observe the effects. It could be done, in a similar way that was done for the links, but time did not permit this to be included in this thesis, and is another suggestion for follow-on research.

C. DOES THE TRANSPUTER ACHIEVE LINEAR PERFORMANCE IMPROVEMENTS?

We could see in Chapter 3 that the four links in one transputer, in some cases gives us linear performance improvements, because the transfer rate per channel is kept constant while we increase the number of channels in parallel. The reader may recall Tables 9, 10 and 11 for 512 bytes or larger.

If we now look into the process performance, turning back to the Test Linearity program, we can say that for this program, each counter took 130 mec to execute and timing from the host process on transputer root we have got a total execution time of 133 msec since the first flag left channels hostout0 to 3, up to the last result was received back.

A simple test was made mapping all processes assigned to a B003 board with four transputers, to only one transputer. In this way, one route process plus 3 counters would run in parallel in only one CPU, the former routers. See Figure 4.10 that shows the new procedure route5 that accomplish that. The configuration now was the same one depicted on Figure 3.14, with a different process placement shown on the program structure on Figure 4.11 . The results obtained are listed on Table 25 .

As we see to have a rigorous linear increase of performance we should have:

1 on each counter time:
   - 517.5 (average)/ 4 = 129.44 , and what we had got was 130 msec each!

2 on the total execution time
   - 534 / 4 = 133.5, and what we had got was 133 msec!

```
PROC route5 (CHAN messagein,messageout,VALUE i)=
  PROC route(CHAN messagein,messageout,routeto1,routeto2,routeto3,
                  echofrom1,echofrom2,echofrom3,VALUE k)=
  ---This procedure is the same of Figure 4.9 and is not repeated.
  PROC counter (CHAN in, out, VALUE tnumber)=
  ---This procedure is the same of Figure 4.7 and is not repeated.

  DEF totlinks = 32:       ---constant for soft channel definitions
  CHAN pipe[totlinks]:     ---soft channel definitions
  PAR
    route (messagein,messageout,pipe[9+(6*i)],pipe[11+(6*i)],
     pipe[13+(6*i)],pipe[8+(6*i)],pipe[10+(6*i)],pipe[12+(6*i)],i)
    counter(pipe[9+(6*i)],pipe[8+(6*i)],((10*i)+1))
    counter(pipe[11+(6*i)],pipe[10+(6*i)],((10*i)+2))
    counter(pipe[13+(6*i)],pipe[12+(6*i)],((10*i)+3)) :
```

Figure 4.10 Procedure Route5.

Another version of the Test Linearity Program was made and mapped to only one transputer T414 in a B003 board. The time for execution was then **2.3 seconds!** A last version made for the OPS system running on the VAX VMS run at best in **8.8 seconds!**

Conclusion 8

With normal communication load, linear increase
of performance with more processors may be achieved!
The routing process does not drag the processor!

```
-- PROGRAM testlinearity
--    ************************************************************
--    *   Title :   Test Performance Linearity
--    *   Version : 3
--    *   Mod :  0
--    *   Author :  Jose Vanni Filho, Lcdr., Brazilian  Navy
--    *   Date :  June, 5th,1987
--    *   Programming Language : OCCAM 1
--    *   Compiler : IMS D 600 - TDS
--    *   Brief Description :  This version of test linearity
--    *     mapped into 5 transputers, shows us the increase
--    *     in time to execute the same processes of version 2
--    *     with the reduction of the number of processors, by
--    *     a factor of 4.
--    ************************************************************

__ SC PROC hostproc (CHAN A,B,C,D,E,F,G,H)
--- This procedure is the same included in version 2 of the Test
--- Performance Linearity program in Appendix F and is not repeated.

-- SC PROC route5 (CHAN messagein,messageout,VALUE i)
  ---This procedure is the same of Figure 4.10 and is not repeated.

-- configuration
-- link definitions
DEF link0in = 4 :
DEF link0out = 0 :
DEF link1in = 5 :
DEF link1out = 1 :
DEF link2in = 6 :
DEF link2out = 2 :
DEF link3in = 7 :
DEF link3out = 3 :

DEF root = 100:
DEF totlinks = 32:
CHAN pipe[totlinks]:

PLACED PAR
  PROCESSOR root
    -- link placements and process assignment
    PLACE pipe[0] AT link0in :
    PLACE pipe[1] AT link0out :
    PLACE pipe[2] AT link1in  :
    PLACE pipe[3] AT link1out :
    PLACE pipe[4] AT link2in  :
    PLACE pipe[5] AT link2out :
    PLACE pipe[6] AT link3in  :
    PLACE pipe[7] AT link3out :

    hostproc (pipe[0],pipe[2],pipe[4],pipe[6],
              pipe[1],pipe[3],pipe[5],pipe[7])

  PLACED PAR j = [0 FOR 4]
    PROCESSOR 10*j
      -- link placements and process assignment
      PLACE pipe[2*j] AT link0out :
      PLACE pipe[(2*j)+1] AT link0in :

      route5 (pipe[(2*j)+1],pipe[2*j],j)
```

Figure 4.11 Structure of Program Test Linearity (5).

TABLE 25

COMPARING COUNTER EXECUTION TIME IN 4 AND 16 TRANSPUTERS NETWORK

	16 transputers	NR	4 transputers	NR
counter 00	130 msec	00	520 msec	00
counter 01	130 msec	01	518 msec	00
counter 02	130 msec	02	517 msec	00
counter 03	130 msec	03	515 msec	00
counter 10	130 msec	10	520 msec	10
counter 11	130 msec	11	519 msec	10
counter 12	130 msec	12	517 msec	10
counter 13	130 msec	13	515 msec	10
counter 20	130 msec	20	520 msec	20
counter 21	130 msec	21	519 msec	20
counter 22	130 msec	22	517 msec	20
counter 23	130 msec	23	515 msec	20
counter 30	130 msec	30	520 msec	30
counter 31	130 msec	31	519 msec	30
counter 32	130 msec	32	517 msec	30
counter 33	130 msec	33	515 msec	30
Total Execution (timed on b001)	133 msec		534 msec	

V. CONCLUSION

When this research begun, in October 1986, we had a new machine, working with a language that we did not know, and using a concept that still today is considered hard to grasp and to work with : Concurrency and Parallelism. After working for eight months with the transputer, the first conclusion that come up is :

- Concurrency and Parallelism are not difficult concepts to understand at all, using the Transputer and the Occam Programming Language.

In this first phase of the research, the evaluation of the Transputer hardware, several significant conclusions were reached and they are summarized in the following paragraphs, that were obtained from the body of the thesis. They give us a good first idea of the real potential and capabilities of the Transputer when programmed in Proto-Occam.

The bit rate in the links is switchable between 10 mbits/sec and 20 mbits/sec. When operating at 10 mbits/sec rate, the data rate was at best 3.8 mbits/sec or 450 kbytes/sec, per channel. So, the eight links will be able, in the best case, to exchange 3.8 mbytes of data in one second, between two adjacent transputers, because the links are really able to operate in parallel. We shall remember that to obtain this results, we need to use the BYTE SLICE or WORD SLICE constructs, with messages larger than 256 bytes. Equally, when switched to 20 mbits /sec rate the maximum data rate obtained was 6.1 mbits/sec.

When a computation bound process is running in the cpu, with the same priority as the routing process, it will reduce the transfer rate on the links for any construct, at least 8% for one channel operating, and 21% for any other number of channels. These results were observed for message size 10,000 bytes or smaller.

On the other hand, if we give high priority to the communications, the cpu process will be executed in the same way, and the communications will keep the previously obtained rate of 3.8 mbits/sec, so this is strongly recommended.

Communications in the links will reduce the performance of a process being executed in the same CPU, when message sizes overcome a threshold size, depending on the process type. For our observed case this value was 520 bytes. For larger message sizes, the maximum reduction in performance for the computation bound process was 50% in the worst case (Six channels operating in parallel).

The transputer is able to increase throughput linearly with the increase of the number of transputers in which the process is executed.

Although very promising, these conclusions are not complete and here follows some suggestions for follow-on work in the evaluation:

1 To investigate the usage of the internal memory by the processor, specially if priority is given for data or program execution code, to be placed in internal memory.

2 To investigate how the scheduler handles long communication processes that are consuming more than one time slice.

3 To use a Logic State Analizer capable to sample in a clock rate of 50 or 100 mhz, to more precisely measure the time delays involved in the receipt of a frame and dispatch of respective acknowledge.

4 To time the amount of time needed for an array of variable size to be transmitted through several transputers to a non adjacent destination.

5 To use the Link Evaluation Program with greater message sizes. This would imply in using B003 boards that have 256 kbytes available, per transputer, instead of the 64 kbytes available at the B001 board, or a replacing the B001 board by another board with larger external memory.

6 To make a thorough study of the effect of link operation over a computation bound process.

7 To benchmark a network of transputers configured in a hypercube with the commercially available hypercube computers, like the Intel IPCS-VX, using the Operating System presented by Cordeiro [Ref. 6].

Another suggestion for research is the development of real-time application programs to observe the behavior of the machine under normal work load situations.

It is important to mention at this point that, as advertised, we could indeed use transputers with different internal clock cycle, communicating with each other with no problems at all.

Equally important is to remember that in all results obtained in this research, we were using bytes or integers, with no floating point operations. So one other recommended topic for investigation, is the link and processor performance evaluation for floating point data. This could be done in two ways:

1 By using software floating point available in Occam 2, or

2 By using the hardware floating point that will be available with the T-800 transputer.

If we could state, our impression about the transputer, the small size, the simplicity and the speed are the things that really stood out.

As a final suggestion, to enlarge the research horizons at the NPS, we recommend the replacement, when possible, of the the B001 board, interfacing with the VAX, that turned out to be a bottleneck for our 160 MIPS capable Transputer System, either in processing speed, or in memory availability.

Occam is a very easy language to use, the fold editor is very powerful and friendly, and the channels are very good elements for synchronizing processes. But as soon as the Ada compiler becomes available, the research should follow that way and then, comparison with the previously obtained performances, will be helpful in judging the applicability of the Transputer in military real-time systems.

APPENDIX A
LEARNING SEQUENCE

a. **How to Log in**

The first thing one will need is an account on the VAX-VMS to use OCCAM.

There is a group account username "OCCAM" , and through the C.S. Department staff one can get a sub-account to it.

Once a person gets a sub-account, one shall have a password and a login name (normally the last name). With this, one should go to a terminal VT 100 or VT 220 (no other terminal will work !), log in, and as soon as the "S" prompt appears, the VAX/VMS System is ready to begin.

If by any chance, the person already has one account in the VAX/VMS system, what he/she may want to do is to work from his own account. That will be possible, but as soon as the S appears and before one tries to use any of the OPS or/and TDS commands one should type either:

- opssetup --> to use the OPS system, or
- tdssetup --> to use the TDS system.

These commands are already included in the login.com file of the OCCAM account and it is a good idea for one to include them in one's login.com file too. Another thing one may need to do is to move to the "dua0:[OCCAM]" directory to copy files and libraries already created and that, certainly will be useful and save time for anybody.

b. **Learning Sequence**

1. Step 1

The first thing one needs to know is how to use the VMS Operating System. One good choice is to run the online tutorial VMSCAI and/or get a VMS tutorial from the C. S. Department [Ref. 22]. If the person is completely unexperienced it will take two sessions of two hours each, to get a good feeling for it.

2. Step 2

When one feels comfortable using the VAX/VMS, the next step is to get acquainted with the fold editor. This is a very powerful editor but most likely it will be new for anybody, and if one needs more information on it, he/she should refer to the Occam Programming System Manual [Ref. 13: section 2].

To execute the tutorial :

- copy from the OCCAM account the file "OPSTUTOR.DOC" using the following commands at the S prompt:
  - "set default dua0:Occam " (this will move you to OCCAM directory)
  - "opscopy opstutor.doc [.your_directory]"
  - "set default [.your directory] (to move back)
- type : " ops opstutor.doc" at the S prompt in your directory

This will open the opstutor.doc file and will appear on the screen on the upper left

"Press -ENTER FOLD- to start session"

"...F OPSTUTOR.DOC"

At this point one should press the key "0" and while pressing it press also key "7" (both keys are on the numerical keypad on the right side of your keyboard. This is the ENTER FOLD command. From here on just follow the on screen instructions.

It is likely one will need about two hours for the first time, but as one keeps using the editor he/she will find it most easy and powerful. It is a must to have a card with a xerox copy off the keypad description codes. See Figure A.1[11].

3. Step 3

Learning the Occam language is the next thing to do.

One may even begin reading the Occam Programming Manual [Ref. 13,: section 3] or Pountain's book [Ref. 12] early in the learning process, if desired. If the reader knows any other structured language such as PASCAL, ADA, or C it will be most easy. It is very important to get a good grasp of the channel concept!

4. Step 4

At this point it would be good one know some thing about the transputer hardware, and architecture. The Transputer Reference Manual is the reference, but the technical notes from INMOS or the existing theses will also help.

5. Step 5

At this point one have a choice of learning one of the three systems available at the NPS: OPS, TDS for the VAX, or TDS for the PC. They are a little bit different and a good choice for the beginner will be the OPS. This will enable the person to use the Occam language for create concurrent programs, that will be

[11]Reproduced By Permission of INMOS Corporation

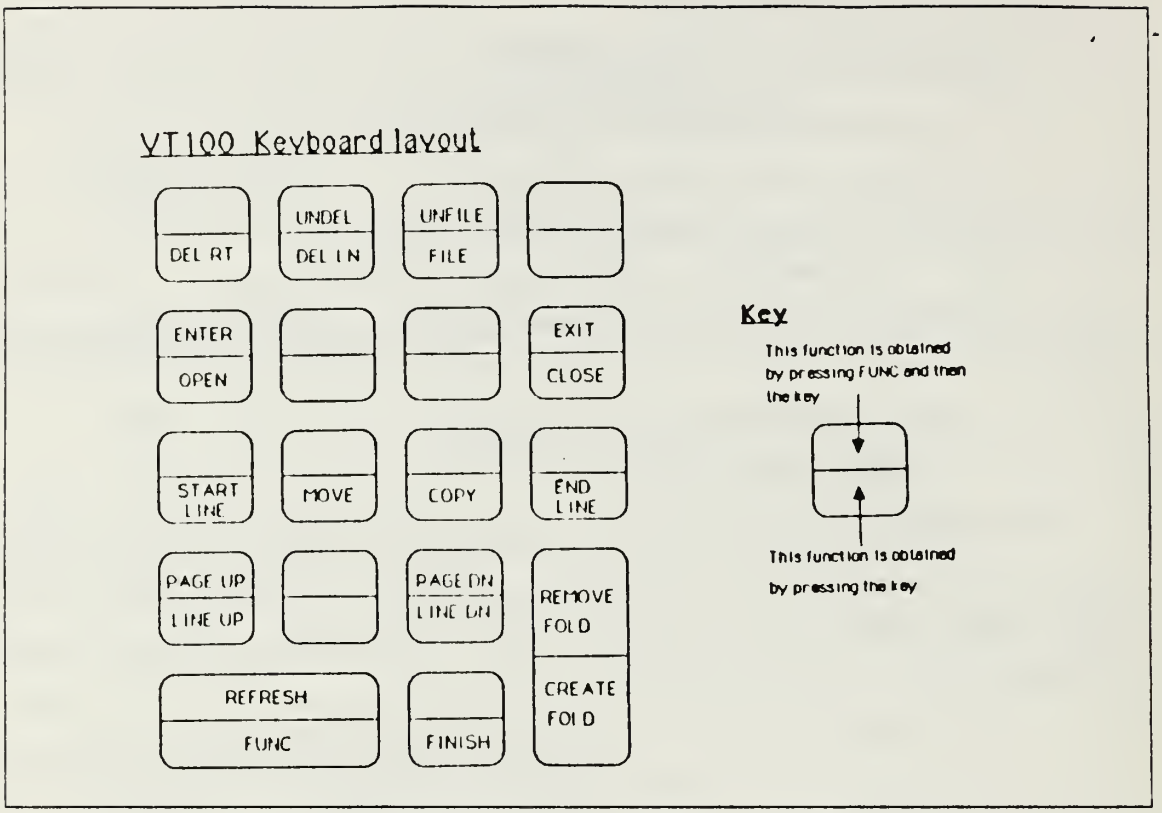


Figure A.1 Keypad for Using the Fold Editor.

compiled, linked, debugged and run on the VAX. The OPS Manual is the main reference for it.

6. Step 6

After that then, depending on which system you will work you should learn the TDS for the VAX or for the PC. The reference manuals respectively are the main reference, but the Theses by Cordeiro or Vanni present several hints and suggestions that may help. With respect to Occam the only different skill one will need is how to make configurations. Again both theses will help.

APPENDIX B
OPS TUTORIAL

1. Introduction

This appendix will describe briefly how to use the OPS system, resident on the VAX/VMS, to write a program, compile, link and execute it. It will not be a complete description of the system and it assumes the reader already knows how to use the Fold Editor and the VMS Operating System in the VAX, and had already been exposed to the Occam Language. The main reference is the Occam Programming System Manual.

2. The Existing File Types:

In OPS there are several user file types identified by the file extension:

- ".ops" - these are source files, folded, that may be edited, and once in the program format, may be compiled. These can not be printed.
- ".lis" - these are listing files that may be used as a VMS file for any purpose. The copy, type, print commands on this operating system work with no problem.
- ".obj" - these are object files that were already compiled. They may be linked to make an .exe file. They are not printable.
- ".exe" - these are executable files that were compiled and linked already. They also can not be printed.

3. To Start the System

Once one is logged on the VAX/VMS on a terminal VT-100 or VT220, the first command to type is:

- opssetup - this will enable all the following commands used in the OPS to be recognized by the VMS Operating System, through the "ops kernel" (opskrnl) resident on the Systems Directory.

4. To Open a File

Type:

- ops "filename" - this command may be applied to any ".ops" file and will make the file available to be edited with the fold editor. Every time one exit the outermost fold, a new version will be created on the VMS file System. Keep track.

5. To Make a Procedure or a Program

The procedures and programs have a similar structure as in Pascal or Ada. After the procedure name with parameters follows the constant, variables and channels declarations and procedures defined only inside that procedure (subprocedures), and at last the main program that may begin with an WHILE, IF, SEQ, ALT, PAR, PRI PAR, a replicator, and so on and terminate with a colon(:). The best is to make all procedures with separate compilation (SC) capability, and for that we should apply the utility MAKE SC PROC to the procedure fold line.

The program has no parameters and no colon at the end, but the structure is the same as described for the procedure. It is important to say that in Occam one is not obliged to declare all constant, variables and channels at the beginning of the procedure. It may be done before any process. A process begins with any of the above mentioned constructs. The best way to learn is to look at ready programs so we will stop this section here. When we use the utility MAKE PROGRAM the name program will automatically appear in front of your program name.

The global_definitions and library are very useful to easily make programs, and it is a good idea to put them in any program.

6. To Compile a Ready Program or Procedure

Any PROGRAM or SC PROC may be compiled separately, as long as the utilities "MAKE PROGRAM" or "MAKE SC PROC" respectively, were applied to them and no error message occurred. To execute the compilation, the cursor has to be in a folded line, with a PROGRAM or and SC PROC inside, and then the user should execute the utility COMPILE. The system will prompt for the object file name and it is a good idea to use the same name of the source file.

7. Debugging a Program During Compilation

The compiler is quick and every time one gets a compilation error, the error description appears on top of the screen and the cursor is placed on the line where the error occurred, or one before. The System will be in edit mode and the error may be corrected at once. After correcting and exiting the fold, one will be ready to compile again, neatly and cleanly.

8. To Link a Program

After the object file was created with the compilation, one has to leave the Fold Editor and at the VMS prompt (S) type:

- link/debug opskrnl, program_name

The debug is optional and we did not use it too much, but we can say it runs and permits one to trace a program execution. After the linking the .exe file will be created, and one is then able to run the program.

9. To Run a Program

After the link was done successfully, one should type after the $ prompt:

- run / debug program_name

Again the debug is optional and after this command the program will be running on the VAX. If logical errors occur, the two options are either to use the VAX on-line debugger or get back to the source code (the .ops file) and place some output to screen (Screen ! var).

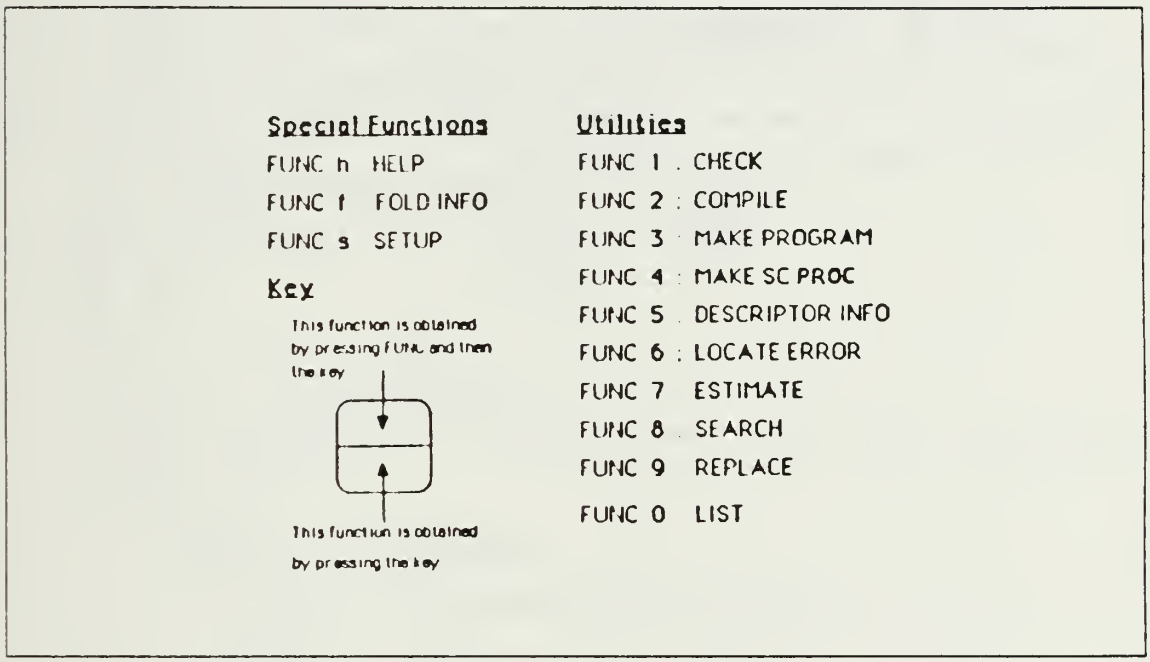


Figure B.1 OPS Utilities.

10. To List a Program

There are two ways to do that:

1 The first one is under VMS, one shall use the OPS command:

- opslist filename.ops filename.lis

This will create a list file on filename.lis to be printed on the VAX on-line printer. Be careful here! If one forgets to put a "filename.lis" in the command, the source file will be transformed in a list file with the wrong termination. And

worse, if by chance one purges the directory, all the fold structure the programer created will be destroyed and will have to be redone, if one needs to compile the program again.

2 The second one is under OPS; one shall use utility LIST. This may be applied to any fold inside the program and the user will be prompted for a file name.lis .

It is important to mention that every time one lists a file, the folds will be opened, and appear sequentially. It is not very easy for a begginer to follow a printout of the file. The fold editor permits us a much better block view of what the program looks like. So most likely if one has a very hard bug to solve, debugging from the screen will be easier.

11. Final Remmarks

There are other commands and utilities that after a while one may need to use, but for the very beginning, the ones listed here will suffice. Figure B.1 show all OPS Utilities and how to call them, on a VT-100 Terminal. The FUNC means that one should press the 0 key at the numerical keypad and the number on the keyboard (NOT PF KEYS). Figure &firstpro presents a simple program as an example.

```
-- PROGRAM hello1
-- hello1
--- ************************************************************
--- This is a the first program in OPS to be seen by a beginner.
--- This fold contains a simple occam program which says hello.
--- After the message appears on the screen you can type any
--- character. --- It will be echoed on the screen (no automatic
--- line feed or carriage --- return.
--- When you type "0" the program ends.
--- ************************************************************

-- declarations
DEF  hello = "hello! press 0 to stop running":
DEF  EndBuffer = -3:    --- system's constant
CHAN Screen AT 1:       --- system's channel
CHAN Keyboard AT 2:     --- system's channel
VAR  ch:
VAR  going:             --- Boolean

-- main program
SEQ
  SEQ i = [1 FOR hello[BYTE 0]]
    Screen ! hello[BYTE i]
  Screen ! EndBuffer  --- EndBuffer needed when outputting strings
  going := TRUE
  WHILE going
    SEQ
      Keyboard ? ch
      Screen ! ch ;EndBuffer
      IF
        ch = #30        --- Hex value for ASCII 0
          going := FALSE
        TRUE
          SKIP
```

Figure B.2 First Program in OPS.

APPENDIX C
TDS TUTORIAL

1. Introduction

This appendix will describe briefly how to use the TDS system, resident on the VAX/VMS at the NPS, to edit, compile, down load and execute an Occam program. It will not be a complete description of the system and it assumes the reader already knows how to use the OPS System, the Fold Editor and the VMS Operating System in the VAX. The main reference is the Transputer Development System Manual, D-600.

2. The Existing File Types:

In TDS there are several user file types identified by the file extension:

- ".tds" - these are source files, folded, that may be edited, and once in the program format, may be compiled. These can not be printed.
- ".lst" - these are listing files that may be used as a VMS file for any purpose. The copy, type, print commands on this operating system work with no problem. Originally the extension was ".lis", but we suggest the programmer to use other termination in order to identify the file.
- ".tcd" - these are "transputer code" files originated from an extraction after a compilation was successfully completed. They are not printable.
- ".cde" - these are non- executable files that were compiled and extracted already. They will exist when the programmer uses closed files inside his program, and contain the code for a file. They are not printable.
- ".dsc" - these are descriptor files and will exist only when the programmer used closed files in his programs. They are not printable.

3. To Start the System

Once one is logged on the VAX/VMS on a terminal VT-100 or VT220, the first command to type is:

- tdssetup - this will enable all the following commands used in the TDS to be recognized by the VMS Operating System, through the "ops kernel" (opskrnl) resident on the Systems Directory.

4. To Open a File

Type:

- tds "filename" - this command is to be applied to any ".tds" file and will make the file available for editing with the fold editor. Every time you exit the outermost fold, a new version will be created on the VMS file System. Keep track.

5. To Make a Procedure or a Program

The procedures and programs have a similar structure as in OPS, so they will not be repeated here.

The global_definitions and library are very useful to make programs easily, and it is a good idea to put them in any program.There are two global_definitions, one for each of the systems identified by the extension. Be careful to imbed in your program the "global_def.tds."

There are two different things from OPS in a program for the TDS:

1 The first is: to see any result on the screen, one must include inside the program the terminal driver, provided by INMOS, for the board that one is using (B001, B002 or B004).

2 The second is the need for a configuration. The configuration basically gives names to the physical channels and places in each transputer the process to be executed there. Rather than try to explain here, the best is to browse some of the several configurations existing in the Theses by Vanni or Cordeiro, or in the programs already existing in the Group account Occam.

6. To Compile a Ready Program or Procedure

Any PROGRAM or SC PROC may be compiled separately as long as the utilities "MAKE PROGRAM" or "MAKE SC PROC" respectively, are applied to them and generate no error message. To execute the compilation, the cursor has to be in a folded line with a PROGRAM or and SC PROC inside, and apply the utility COMPILE. There will be no prompt at this time, except for the compilation parameters. If the program has complicated nesting of PAR and ALT constructs, use CHECK = FALSE.

7. Debugging a Program During Compilation

The compiler is quick and every time one gets a compilation error, the error description appears on top of the screen and the cursor will be placed on the line where the error occurred, or one before, in edit mode and the error may be corrected at once. After corrected, exit the fold and one will be ready to compile again. Neat and Clean.

8. To Extract the Code to Be Executed in the Transputer

The compilation will create several folds inside the .ops program containing the descriptor and the code to be executed. To extract the code execute utility " EXTRACT TO FILE ". At this point one will be prompted for a filename to extract, and we strongly suggest to use the same name of the source file.

9. To Down Load and Run a Program

Once the ".tcd" file was created, the user will be ready to run the program on the transputer network. Before you down load, check the wiring diagram (Utility 7), and see if the links are properly connected. After this, exit from the fold editor and execute at the VMS (S) prompt :

TDSLOAD filename.tcd

What will happen is that the file will be opened by the VAX and the programmer will be prompted for the escape sequence (normally is ESC ESC ESC). After typing the escape sequence the transputer become active and the code is loaded. Check the Manual if any Error message occurs. After the program is down loaded, it will be executed at once, with no need of any other intervention of the user. To stop the transputer press reset at the B001 board.

10. To List a Program

There are two ways to do that:

1 The first one is under VMS, one shall use the OPS command:

- opslist filename.ops filename.lst

This will create a list file under filename.lst to be printed at the VAX on-line printer. Be careful here! If one forgets to put a "filename.lst" on the command, the source file will be transformed in a list file with the wrong termination. And worse if by chance one purges the directory, all the fold structure the programmer created will be destroyed and will have to be redone, if one needs to compile the program again.

2 The second one is under TDS; one shall use utility LIST. This may be applied to any fold inside the program and the user will be prompted for a "filename.lis". We suggest the termination to be changed to .lst to differentiate from the OPS list files.

11. Final Remarks

There are other commands and utilities that after a while one may need to use, but for the very beginning, the ones listed here will suffice. Figure C.1 shows the utilities for the TDS System.

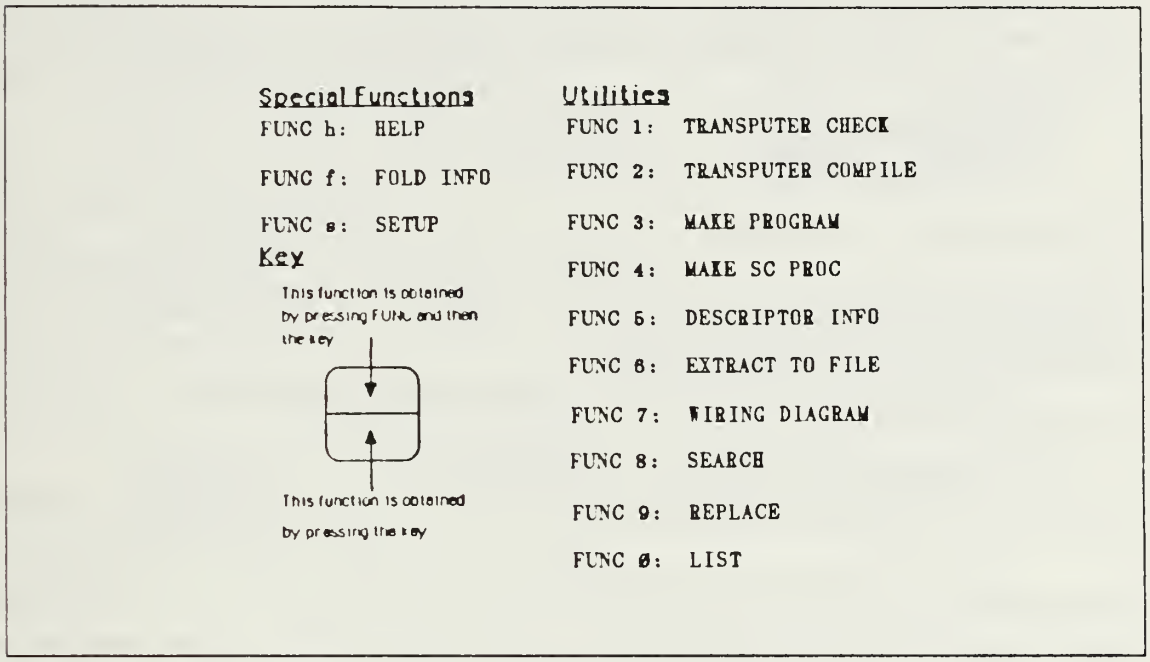


Figure C.1 The Utilities for the TDS System.

APPENDIX D
HINTS ABOUT OCCAM PROGRAMMING

The goal of this appendix is two fold. First to mention some different and interesting facts that happened to us and may happen to anyone programming for the first time in Occam, and second to make some comments about the Link Evaluation Program.

a. **Program Structure**

The program structure for OPS and TDS is quite similar, just differing in the global definitions , configuration, and some predefined procedures. The difference in the global definitions is a very critical one. While in the OPS we place the CHAN Screen AT 1 and CHAN Keyboard AT 2, in TDS we just declare CHAN Screen: and CHAN Keyboard:, because the Screen and Keyboard handling is done throughout the terminal_driver.

The configuration section of a program is the one were we map the physical channels and the processes onto the processors, and it only exists for the TDS system.

The pre-defined run time procedures are described in detail in the TDS manual and the OPS manual, but they only can be used with the TDS. Some examples are: the BYTE.SLICE.INPUT, PUT.BYTE, READ.BYTE, WORD.SLICE.OUTPUT, etc..

1. A program in OPS

Figure D.1 describes the structure of an OPS program.

```
PROGRAM  progname
  global_def.ops (collection of system defined constants)
  library.occ (if wanted)
--- any procedures used inside your program   (optional)
  SC PROC 1 --- any separate compilation procedure that one may
  SC PROC 2 --- refer to and call from the main of the program
  PROC 3  (parameters...,.. )  ---procedures called by the main.
--- local definitions for the main
--- main
  SEQ --- it could be PAR, ALT, WHILE TRUE, IF or a replicator
    code
```

Figure D.1 OPS Program Structure.

2. A Program in TDS

Figure D.2 describes the structure of a TDS program showing as an example the structure of the LINK EVALUATION PROGRAM.

```
PROGRAM  link.evaluation
--- each one of the following procedures have the same structure as
--- depicted on figure D.1
  SC PROC hostproc (parameters)        --- code for transputer root
  SC PROC transfer0.B003 (parameters) --- code for transputer0
  SC PROC transfer1.B003 (parameters) --- code for transputer1
  SC PROC transfer2_B003 (parameters) --- code for transputer2
  SC PROC transfer3_B003 (parameters) --- code for transputer3

  --- configuration
  ... link definitions
  ... physical channels declaration
  PLACED PAR
    PROCESSOR ROOT  --- ROOT = 100 (one may use any process number)
      ...channel placements (physical placement of the channels
                              (according the network topology)
      hostproc (physical channel parameters)
       --- the process hostproc is the outermost placed on
       --- transputer root and has to be an SC PROC
    PROCESSOR 0 --- Like shown for transputer root, in each of
    PROCESSOR 1 --- the processors it is made a physical channel
    PROCESSOR 2 --- placement and  a process placement.
    PROCESSOR 3 ---
```

Figure D.2 TDS Program Structure Example.

The two Figures D.1 and D.2 give to the reader an idea of the general structure of an OPS and a TDS program. Normally, the terminal driver is one of the SC PROCS, inside the process placed in the transputer root, to permit user and screen interaction. Cordeiro [Ref. 6], describes in detail how to make a configuration and how to map a program made for OPS into the TDS system, and therefore it will not be addressed here. Again, the best way to begin programming in Occam is to look at sample programs already made.

b. **Problems and Suggestions**

1. Setting up Some Standards

Early in the learning process we felt necessary to standardize some of our procedures when programming. This may be not the best, but this is what we came out after several changes through the research process, and is given as a suggestion only:

- Use all your procedure and variable names in lower case. The system has some predefined variables like "EndBuffer", and all reserved words are uppercase. So

doing this, one will not have problems of naming because both the OPS and TDS are case sensitive. For example you may use a variable named "true" and no problem with the system defined "TRUE" will occur.

- When in the code one has a replicator with multiple statements under it use always a SKIP as shown in Figure D.3 . That will make certain that the last index value is executed.
- In programs with repetitive interactions with the user, use a new.line after each execution and before the new prompt to the user
  (Keyboard ? var). This will prevent unwanted multiple executions.
- Every time a comment is placed in the code, use at least 3 dashes. This will enable one to recognize easily in the printout, what is comment, and what is the beginning of a fold.

```
SEQ i = [0 FOR 5]
  SEQ
    in ? var1
    out ! var1 + 1
SKIP   --- this is the SKIP we felt necessary
```

Figure D.3 SKIP Usage.

2. When Making Any Procedure

In order to permit any procedure to run in parallel (always), with any other process, use as much channels as possible as parameters, instead of VAR or VALUES. The channels will enable the programmer to exchange data between two procedures without a procedure call. This is the key for the parallelism. One good example were this was used is the procedure cpubusysum, in Appendix E. Other examples can be seen in the library routines defined inside the procedure getchoice, also in Appendix E.

Also make the procedures, SC PROCs, as much as possible. This is better for the programmer because if an compilation error occurs, it will be detected earlier and the recompilation time will be shorter. It is also better for the compiler because it stays away from the compilation limit.

3. When Compiling

When compiling, several errors may be flagged. If an error message:

- "... shared variable varname" , occurs, change the check compilation parameter to false. When check is true even the output of the same variable to several different channels in parallel, will make the compiler flag the error, when it does not exist really.

If any errors occur, the compiler will position the cursor always before the error exact position. Sometimes the error will be on the same line, and sometimes in the next line of code.

4. When Making Large Programs

When making large programs, one should take care of the compiler code limit either for OPS or TDS. In the VAX this limit is around 100 blocks, or 50 kbytes of code. To get around this problem, one should make some procedures inside the program as Separate Compilation (SC) procedures and the compiler than will be able to handle it.

5. When Down Loading the Code

When down loading the code, several times a message like the following one will occur:

- "... Illegal board function" - we had that a lot with no reasonable cause. The action taken when this happened was to down load again, sometimes up to 4 times to have the code down loaded properly to the transputer network.

c. Comments About the Link Evaluation Program

The Link Evaluation Program takes about 340 blocks of the VAX, or approximately 170k of code and comments.

Our approach in doing the Link Evaluation Program was Top down and we think it this was the right one. First the general structure was made, with all procedures but the user interface and the terminal driver replaced by stubs. When this was running, then one by one the byte.slice.transfer, the inout.transfer, the word.slice.transfer and finally the int.transfer were added. Even though all these procedures where pre-tested using dedicated harnesses, some times new bugs came out as they were put together.

In general the structure of the program is based on the four procedures just mentioned, that reside one of each, in each of the transputers. When executed,the the user choice of construct make the respective procedure be executed in parallel in all 5 transputers.

1. Most Common Errors

- Bad definition of buffer limits and lack of initialization.
- mismatch of channel usage - a process outputting to a channel that no other process was waiting for an input.
- compilation limit achieved - this happened in procedure hostproc and in order not to affect the performance measurements, the SC procedure get.choice was

created using part of the user.interface code, and so procedure user.interface passed to call getchoice.

The difficulty of finding the first two problems is due to the symptom to be the program freezing in execution on the screen and no message coming. To find were the error was occurring approximately, we placed some "Screen ! var" statements in the middle of the code, and from then on only reading the code and guessing what it could be, worked. We tried, and succeeded, also to trace the execution, by looking at the listed code and following the flow of communications.

As a final comment, the facility to reuse previously created software is tremendous. Each configuration just need to be done once, and can be always reused by just changing the name of the placed procedures. The procedures and programs can be annexed to a new file or filed with one key stroke, the utility file/unfile of the fold editor.

APPENDIX E
THE LINK EVALUATION PROGRAM

```
-- header.occ
---  ***************************************************************
---  *   Title    :  Link Evaluation Program                        *
---  *   Version  :  7                                              *
---  *   Mod      :  0                                              *
---  *   Author   :  Jose Vanni Filho, Lcdr., Brazilian  Navy       *
---  *   Date     :  June / 02 / 1987                               *
---  *   Programming Language : OCCAM 1                             *
---  *   Compiler :  IMS D 600 - TDS                                *
---  *   Purpose  :  To Evaluate the Transputer link transfer rate  *
---  *               for several channel parallelism situations,    *
---  *               construct types, and different cpu loads       *
---  ***************************************************************

-- Brief description of program
---  ***************************************************************
---  Interactive program that uses the INMOS links at 10 Mbits/sec and
---  evaluates the transfer rates from the b001 board to the b003 board
---  using one to four channels in parallel for output and input.
---  The program calculates and display the transfer rate after a
---  specified number of runs (20 for now) in a table format for
---  the following block.size and channel configurations:

-- Block Sizes
---        1  -   2  -   4  -    8 -   16 -   32 -   64 -   128
---      256 - 512 - 1024  - 1280 - 2048 - 4096 - 8192 - 10000

-- Channel configurations
---   1 out    - 1 channel(output) in one link
---   1 in/out - 2 channels(input and output) in par in one link
---   2 out    - 2 channels(output) in parallel in two links
---   2 in/out - 4 channels(input and output) in par in two links
---   3 out    - 3 channels(output) in parallel in three links
---   3 in/out - 6 channels(input and output) in par in three links
---   4 out    - 4 channels(output) in parallel in four links
---   4 in     - 4 channels(input) in parallel in four links
---   4 in/out - 8 channels(input and output) in par in four links

-- User options during program execution
---  User Options:
---  CPUs MODEs OF OPERATION
---    0 - No concurrent process in the cpus
---    1 - B003 cpus with sum process concurrently (par)
---    2 - all cpus with sum process concurrently (par)
---    3 - B003 cpus with sum process concurrently (pripar)
---    4 - all cpus with sum process concurrently (pripar)
---    5 - B003 cpus with array product process concurrently (par)
---    6 - all cpus with array product process concurrently (par)
---    7 - B003 cpus with array product process concurrently(pripar)
---    8 - all cpus with array product process concurrently (pripar)
---
---  CONSTRUCTS AND DATA TYPES
---    A - input/output channels   ( CHARACTERS (BYTES))
---    B - byte slice input/output ( CHARACTERS (BYTES))
---    I - input/output channels   ( INTEGERS   (WORDS))
---    W - word slice input/output ( INTEGERS   (WORDS))
---  ***************************************************************
```

```
---  ************************************************************************
-- PROGRAM link.evaluation
---  ************************************************************************
-- link.evaluation PROCESSES
-- TRANSPUTER_ROOTB001.TDS
-- SC PROC hostproc
-- PROC hostproc (CHAN A,B,C,D,E,F,G,H)
PROC hostproc (CHAN A,B,C,D,E,F,G,H) =
-- description
--- ************************************************************************
--- This is the outer procedure placed on transputer Root. It contains
--- global variables and constants, and all procedures that run in this
--- transputer. It executes in parallel the procedures :
---                terminal.driver and user.interface
--- ************************************************************************
  -- global_def.tds (partial)
  -- Constants Definitions
  DEF EndBuffer = -3:
  DEF port      =  0:--- assign the i/o port of the B001 to terminal
  DEF baud      = 11:--- set the baud.rate to 9600 bps
  --- constantly used ASCII values
  DEF tab       =  9:
  DEF lf        = 10:
  DEF cr        = 13:
  DEF esc       = 27:
  DEF sp        = 32:
  -- Channels Definitions
  CHAN Screen   :  --- defined for output to the Screen
  CHAN Keyboard :  --- defined for input from the Keyboard
  -- Link Definitions
  DEF link0out = 0 :
  DEF link1out = 1 :
  DEF link2out = 2 :
  DEF link3out = 3 :
  DEF link0in  = 4 :
  DEF link1in  = 5 :
  DEF link2in  = 6 :
  DEF link3in  = 7 :
  -- library.occ  (partial)
  -- io_routines.occ
  -- PROC new.line
  PROC new.line =
  ---  ***********************************************************************
  ---  jumps to a new line on the screen
  ---  ***********************************************************************
    SEQ
      Screen ! cr;lf;EndBuffer :
  -- PROC write.string (VALUE string[])
  PROC write.string (VALUE string[]) =
  --- ***********************************************************************
  --- Writes a given string to the screen, in a byte by byte fashion  *
  --- ***********************************************************************
    SEQ
      SEQ i = [1 FOR string[BYTE 0]]
        Screen ! string[BYTE i]
      Screen ! EndBuffer :
  -- PROC clear.screen
  PROC clear.screen =
  --- ***********************************************************************
  --- clears the screen
  --- ***********************************************************************
    SEQ
      Screen ! esc;'[';'2';'J';EndBuffer --- clear screen sequence
      Screen ! esc;'[';'H' :             --- home cursor
  -- PROC write.number (VALUE number)
```

```
--- ************************************************************
--- This PROC outputs a signed integer value to the screen   *
--- ************************************************************

PROC write.number(VALUE number) =
  VAR output[16], count, x:
  SEQ
    x:= number
    count:= 0
    IF
      --  handle special cases
      x=0
        Screen ! '0'
      x<0
        SEQ
          Screen ! '-'
          x:=-x
      TRUE
        SKIP
    WHILE x>0
      --  construct number
      SEQ
        output[count] := (x   10) + '0'
        count := count + 1
        x:= x/10
    WHILE count > 0
      --  output number
      SEQ
        count := count-1
        Screen ! output[count]
    SKIP:
-- utilities.occ
-- PROC transfer.rate (VALUE start,stop,board.type,nr.of.bytes...)
PROC transfer.rate (VALUE start, stop, board.type, nr.of.bytes,
                    VAR rate) =
  --- ************************************************************
  --- receives two tick values "start" and "stop", number of bytes  *
  --- and board type and outputs the transfer rate.                 *
  --- ************************************************************

  -- board number definitions
  --- board.type = 0 ---->  VAX VMS
  --- board.type = 1 ---->  B001
  --- board.type = 2 ---->  B002
  --- board.type = 31---->  B003 ( high priority )
  --- board.type = 32---->  B003 ( low priority )
  --- board.type = 4 ---->  B004
  --- outputs to the screen the transfer rate in kbits per second

  -- constant definitions
  DEF vax.sec    =10000000 :      --- hundreds of nsec/second
  DEF b001.sec   =  625000 :      --- # of 1.6 microsec/second
  DEF b003h.sec  = 1000000 :      --- # of microsec/second
  DEF b003l.sec  =   15625 :      --- # of 64 microsec/second
  DEF max.number.of.ticks = 2147483648 : --- maximum integer (2**31)

  -- variable declarations
  VAR elapsed.tick :
  VAR factor :       --- to convert ticks to seconds


  SEQ
    elapsed.tick := stop - start
    IF
      elapsed.tick < 0
        elapsed.tick := elapsed.tick + max.number.of.ticks
      TRUE
        SKIP
    -- selection of correct factor in accordance with the board
    IF
```

```
        board.type = 0                  --- VAX VMS
          factor := vax.sec
        board.type  = 1                 --- B001
          factor := b001.sec
        board.type  = 2                 --- B002
          SKIP                           --- to be implemented in the future
        board.type  = 31                --- B003 in high priority
          factor := b003h.sec
        board.type  = 32                --- B003 in low priority
          factor := b0031.sec
        board.type  = 4                 --- B004
          SKIP                           --- to be implemented in the future
      -- rate calculation
      IF
        board.type = 32
          rate := ((nr.of.bytes*8)*factor)/(elapsed.tick*1000)
          --- operation is done this way to keep precision ok!
        TRUE
          rate := ((nr.of.bytes*8)*(factor/1000))/elapsed.tick
          --- operation is done in this way in order to don't exceed
          --- maxint on the numerator.
        --- multiply by 8 due to 8 bits per byte
        --- divide by 1000 to have the transfer rate in kbits/sec
      SKIP:
-- PROC capitalize (VAR ch)
PROC capitalize (VAR ch) =
--- ****************************************************************
--- capitalizes any lower case character into upper case
--- ****************************************************************
  DEF delta =('a' - 'A') :
                                  ---  A ---> 65
                                  ---  a ---> 97      ASCII values
                                  ---  z ---> 122
  SEQ
    IF
      (ch <= 'z') AND (ch >= 'a')
        ch := ch - delta
      TRUE
        SKIP :
-- SC PROC IMS.B001.terminal.driver()
-- TERMINAL_DRIVER.TDS
-- PROC IMS.B001.terminal.driver(CHAN Keyboard,Screen,
                                    VALUE port,baud.rate)
---  ****************************************************************
---  The terminal driver used is the one provided by the
---  manufacturer for the board B001, and for that reason
---  is not included here.
---  ****************************************************************
```

```
-- SC PROC cpubusysum (CHAN flag1,counterchan)    ---- sum
-- CPUBUSYSUM.TDS
-- PROC cpubusysum (CHAN flag1,counterchan)
PROC cpubusysum (CHAN flag1,counterchan)=
-- description
---  ***************************************************************
---  It keeps the cpu working in parallel(time sharing) with the link
---  transfers by doing sum operations . It stops when it receives
---  a flag by the channel flag1 from the transfer procedure that is
---  being executed concurrently. It outputs by channel counterchan
---  the number of operations done.
---  ***************************************************************

  VAR a,b,e,
      working,
      counter,
      ch :
  SEQ
    counter := 0
    working := TRUE
    TIME ? a
    WHILE working
      ALT
        flag1 ? ch
          working := FALSE
        TIME ? b
          SEQ
            e := a + b
            counter := counter + 1
    counterchan ! counter:
-- CPUBUSYSUM.dsc   descriptor
-- CPUBUSYSUM.cde   code
-- SC PROC cpubusyprod (CHAN flag1,counterchan)   ---- product
-- CPUBUSYPROD.TDS
-- PROC cpubusyprod (CHAN flag1,counterchan)
PROC cpubusyprod (CHAN flag1,counterchan)=
-- description
---  ***************************************************************
---  It keeps the cpu working in parallel(time sharing) with the link
---  transfers by doing array multiplications. It stops when receives
---  a flag by the channel flag1 from the transfer procedure, that is
---  being executed concurrently. It outputs by channel counterchan
---  the number of operations done.
---  ***************************************************************
  -- constants and variable declarations
  DEF number = 100:   ---- size of array
  VAR a[number + 1],  ---- array of integers
      b[number + 1],  ---- array of integers
      e[number + 1],  ---- array of integers
      clock,          ---- integer -variable to get time
      working,        ---- boolean -to stop execution
      counter,        ---- integer -number of operations done
      ch :

  SEQ
    -- initialize buffers and variables
    SEQ i = [ 1 FOR number ]
      SEQ
        a[i] := 3*i
        b[i] := 5*i
    SKIP
    counter := 0
    working := TRUE
    WHILE working
      ALT
        flag1 ? ch
```

```
          working := FALSE
        TIME ? clock
          SEQ
            SEQ i = [1 FOR number]
              e[i] := a[i] * b[i]
            counter := counter + number ---updates nr. of operations
    counterchan ! counter:

-- global constant and variable declarations for transputer root
DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                        1024, 1280, 2048, 4096, 8192, 10000 ]:
DEF nr.of.sizes = 16:       --- as counted from above table
DEF maxblock.size = 10000:  --- last from the above table
DEF repetition = 20:        --- for averaging purposes
DEF maxwordblock.size = maxblock.size/4:

CHAN hostin0  AT link0in:
CHAN hostin1  AT link1in:
CHAN hostin2  AT link2in:
CHAN hostin3  AT link3in:
CHAN hostout0 AT link0out:
CHAN hostout1 AT link1out:
CHAN hostout2 AT link2out:
CHAN hostout3 AT link3out:
```

```
-- PROC inout.transfer (VALUE repetition,cpumode)
PROC inout.transfer ( VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- It initializes the buffers and it executes the procedure
--- iotransfer,  and, when applicable one of the following:
---     cpubusy.prod or cpubusy.sum. (according to cpumode)
--- Uses global constant maxblock.size.
--- ****************************************************************

  -- variable declarations
  CHAN flag,    --- flags the cpu to stop
       counter: --- return the number of operations cpu did
  VAR buffer0 [BYTE maxblock.size + 1],
      buffer1 [BYTE maxblock.size + 1],
      buffer2 [BYTE maxblock.size + 1],
      buffer3 [BYTE maxblock.size + 1]:


  -- PROC iotransfer (VALUE repetition, cpumode, CHAN flag, ...)
  PROC iotransfer (VALUE repetition,cpumode,CHAN flag, counter)=
  -- Description
  ---  ************************************************************
  ---  Executes sequentially several parallel transfers of bytes
  ---  to/from one to four transputers using the input/output
  ---  primitive and output to the screen the transfer rate
  ---  values of the output TABLE.
  ---  Uses global constants : sizetable, nr.of.sizes, repetition
  ---  ************************************************************

    -- variable declarations
    VAR block.size,
        actual.rate,
        rate,
        number,       --- the number of operations cpu did
        ch[4],
        deadtime, deadtime0, deadtime1, ----  to calculate deadtime
        time0[4],
        time1[4]:

    SEQ
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          -- making the table
          block.size := sizetable[i]
          write.number (block.size)
          Screen ! tab
          -- calculation of deadtime
          TIME ? deadtime0
          SEQ i = [1 FOR block.size]
            SKIP
          TIME ? deadtime1
          deadtime := deadtime1 - deadtime0
          -- output to one channel
          actual.rate := 0
          SEQ j = [1 FOR repetition]
            SEQ
              hostin0 ? ch[0]
              TIME ? time0[0]
              SEQ k = [1 FOR block.size]
                hostout0 ! buffer0 [BYTE k]
              TIME ? time1[0]
              time1[0] := time1[0] - deadtime
              transfer.rate (time0[0],time1[0],1,block.size,rate)
              actual.rate := ((actual.rate * (j-i)) + rate)/j
          SKIP
          write.number (actual.rate)
```

```
Screen ! tab
-- output/input from one channel
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    hostin0 ? ch[0]
    TIME ? time0[0]
    SEQ k = [1 FOR block.size]
      PAR
        hostout0 ! buffer0 [BYTE k]
        hostin0 ? buffer1 [BYTE k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- output to two channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
    TIME ? time0[0]
    SEQ k = [1 FOR block.size]
      PAR
        hostout0 ! buffer0 [BYTE k]
        hostout1 ! buffer1 [BYTE k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- output/input from two channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
    TIME ? time0[0]
    SEQ k = [ 1 FOR block.size ]
      PAR
        hostout0 ! buffer0 [BYTE k]
        hostout1 ! buffer1 [BYTE k]
        hostin0 ? buffer2 [BYTE k]
        hostin1 ? buffer3 [BYTE k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- output to three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
    TIME ? time0[0]
    SEQ k = [1 FOR block.size]
```

```
        PAR
          hostout0 ! buffer0 [BYTE k]
          hostout1 ! buffer1 [BYTE k]
          hostout2 ! buffer2 [BYTE k]
      TIME ? time1[0]
      time1[0] := time1[0] - deadtime
      transfer.rate(time0[0],time1[0],1,block.size,rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output/input from three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
    TIME ? time0[0]
    SEQ k = [ 1 FOR block.size ]
      PAR
        hostout0 ! buffer0 [BYTE k]
        hostout1 ! buffer1 [BYTE k]
        hostout2 ! buffer2 [BYTE k]
        hostin0 ? buffer0 [BYTE k]
        hostin1 ? buffer1 [BYTE k]
        hostin2 ? buffer2 [BYTE k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output to four channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
      hostin3 ? ch[3]
    TIME ? time0[0]
    SEQ k = [1 FOR block.size]
      PAR
        hostout0 ! buffer0 [BYTE k]
        hostout1 ! buffer1 [BYTE k]
        hostout2 ! buffer2 [BYTE k]
        hostout3 ! buffer3 [BYTE k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- input from four channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
      hostin3 ? ch[3]
    TIME ? time0[0]
```

```
            SEQ k = [ 1 FOR block.size ]
              PAR
                hostin0 ? buffer0 [BYTE k]
                hostin1 ? buffer1 [BYTE k]
                hostin2 ? buffer2 [BYTE k]
                hostin3 ? buffer3 [BYTE k]
            TIME ? time1[0]
            time1[0] := time1[0] - deadtime
            transfer.rate(time0[0],time1[0],1,block.size,rate)
            actual.rate := ((actual.rate * (j-1)) + rate)/j
        SKIP
        write.number (actual.rate)
        Screen ! tab
        -- all output and input in parallel
        actual.rate := 0
        SEQ j = [1 FOR repetition]
          SEQ
            PAR
              hostin0 ? ch[0]
              hostin1 ? ch[1]
              hostin2 ? ch[2]
              hostin3 ? ch[3]
            TIME ? time0[0]
            PAR
              SEQ k = [1 FOR block.size]
                PAR
                  hostout0 ! buffer0 [BYTE k]
                  hostout1 ! buffer1 [BYTE k]
                  hostout2 ! buffer2 [BYTE k]
                  hostout3 ! buffer3 [BYTE k]
              SEQ k = [1 FOR block.size]
                PAR
                  hostin0 ? buffer0 [BYTE k]
                  hostin1 ? buffer1 [BYTE k]
                  hostin2 ? buffer2 [BYTE k]
                  hostin3 ? buffer3 [BYTE k]
            TIME ? time1[0]
            time1[0] := time1[0] - deadtime
            transfer.rate(time0[0],time1[0],1,block.size,rate)
            actual.rate := ((actual.rate * (j-1)) + rate)/j
        SKIP
        write.number (actual.rate)
        new.line
    SKIP
    new.line
    -- send to screen operations done concurrently
    IF
      cpumode = '0'
        write.string (" No other process running concurrently ")
      (((cpumode='2')OR(cpumode='4')) OR
                              ((cpumode='6')OR(cpumode='8')))
        SEQ
          flag ! 'a'
          counter ? number
          write.string ("Number of operations (in //) at ")
          write.string ("the b001 transputer ")
          write.number (number)
          new.line
          hostin0 ? number
          write.string ("Number of operations (in //) at ")
          write.string ("transputer 0 (b003) ")
          write.number (number)
      TRUE
        SEQ
          hostin0 ? number
          write.string ("Number of operations (in //) ")
          write.string ("transputer 0(b003)")
          write.number (number)
```

```
      new.line
      new.line    :

SEQ   --- main inout.transfer
  -- initializing buffers
  SEQ k = [1 FOR maxblock.size]
    SEQ
      buffer0 [BYTE k] := '0'
      buffer1 [BYTE k] := '1'
      buffer2 [BYTE k] := '2'
      buffer3 [BYTE k] := '3'
  SKIP
  IF
    cpumode = '2'
      PAR
        iotransfer (repetition, cpumode, flag, counter)
        cpubusysum (flag, counter)
    cpumode = '4'
      PRI PAR
        iotransfer (repetition, cpumode, flag, counter)
        cpubusysum (flag, counter)
    cpumode = '6'
      PAR
        iotransfer (repetition, cpumode, flag, counter)
        cpubusyprod (flag, counter)
    cpumode = '8'
      PRI PAR
        iotransfer (repetition, cpumode, flag, counter)
        cpubusyprod (flag, counter)
    TRUE
      iotransfer (repetition, cpumode, flag, counter):
```

```
-- PROC byte.slice.transfer (VALUE repetition, cpumode)
PROC byte.slice.transfer (VALUE repetition, cpumode)=
-- description
--- ****************************************************************
--- It initializes the buffers and it executes the procedure
--- transfer,  and, when applicable one of the following:
---     cpubusy.prod or cpubusy.sum. (according to cpumode)
--- Uses global constant maxblock.size.
--- ****************************************************************

  -- variable declarations
  CHAN flag,    --- flags the cpu to stop
       counter: --- return the number of operations cpu did
  VAR buffer0 [BYTE maxblock.size + 1],
      buffer1 [BYTE maxblock.size + 1],
      buffer2 [BYTE maxblock.size + 1],
      buffer3 [BYTE maxblock.size + 1]:

  -- PROC transfer (VALUE repetition, cpumode, CHAN flag, counter)
  PROC transfer (VALUE repetition,cpumode,CHAN flag, counter)=
  -- Description
  ---  ************************************************************
  ---  Executes sequentially several parallel transfers of bytes
  ---  to/from 1 to four transputers using the BYTE.SLICE Procedure
  ---  and output to the screen the transfer rate values of the
  ---  output TABLE.
  ---  Uses global constants : sizetable, nr.of.sizes, repetition
  ---  ************************************************************

    -- variable declarations
    VAR block.size,
        actual.rate,
        rate,
        number,       --- the number of operations cpu did
        ch[4],
        time0[4],
        time1[4]:


    SEQ
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          -- making the table after each io operation
          block.size := sizetable[i]
          write.number (block.size)
          Screen ! tab
          -- output to one channel
          actual.rate := 0
          SEQ j = [1 FOR repetition]
            SEQ
              hostin0 ? ch[0]
              TIME ? time0[0]
              BYTE.SLICE.OUTPUT(hostout0,buffer0,1,block.size)
              TIME ? time1[0]
              transfer.rate (time0[0],time1[0],1,block.size,rate)
              actual.rate := ((actual.rate * (j-1)) + rate)/j
          SKIP
          write.number (actual.rate)
          Screen ! tab
          -- output/input to one channel
          actual.rate := 0
          SEQ j = [1 FOR repetition]
            SEQ
              hostin0 ? ch[0]
              TIME ? time0[0]
              PAR
                BYTE.SLICE.OUTPUT(hostout0,buffer0,1,block.size)
```

```
        BYTE.SLICE.INPUT(hostin0,buffer1,1,block.size)
      TIME ? time1[0]
      transfer.rate (time0[0],time1[0],1,block.size,rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output to two channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
    TIME ? time0[0]
    PAR
      BYTE.SLICE.OUTPUT(hostout0,buffer0,1,block.size)
      BYTE.SLICE.OUTPUT(hostout1,buffer1,1,block.size)
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output/input from two channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
    TIME ? time0[0]
    PAR
      BYTE.SLICE.OUTPUT(hostout0,buffer0,1,block.size)
      BYTE.SLICE.OUTPUT(hostout1,buffer1,1,block.size)
      BYTE.SLICE.INPUT(hostin0,buffer2,1,block.size)
      BYTE.SLICE.INPUT(hostin1,buffer3,1,block.size)
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output to three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
    TIME ? time0[0]
    PAR
      BYTE.SLICE.OUTPUT(hostout0,buffer0,1,block.size)
      BYTE.SLICE.OUTPUT(hostout1,buffer1,1,block.size)
      BYTE.SLICE.OUTPUT(hostout2,buffer2,1,block.size)
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output/input from three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
```

```
        hostin1 ? ch[1]
        hostin2 ? ch[2]
      TIME ? time0[0]
      PAR
        BYTE.SLICE.OUTPUT(hostout0,buffer0,1,block.size)
        BYTE.SLICE.OUTPUT(hostout1,buffer1,1,block.size)
        BYTE.SLICE.OUTPUT(hostout2,buffer2,1,block.size)
        BYTE.SLICE.INPUT(hostin0,buffer0,1,block.size)
        BYTE.SLICE.INPUT(hostin1,buffer1,1,block.size)
        BYTE.SLICE.INPUT(hostin2,buffer2,1,block.size)
      TIME ? time1[0]
      transfer.rate(time0[0],time1[0],1,block.size,rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- output to four channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
      hostin3 ? ch[3]
    TIME ? time0[0]
    PAR
      BYTE.SLICE.OUTPUT(hostout0,buffer0,1,block.size)
      BYTE.SLICE.OUTPUT(hostout1,buffer1,1,block.size)
      BYTE.SLICE.OUTPUT(hostout2,buffer2,1,block.size)
      BYTE.SLICE.OUTPUT(hostout3,buffer3,1,block.size)
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- input from four channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
      hostin3 ? ch[3]
    TIME ? time0[0]
    PAR
      BYTE.SLICE.INPUT(hostin0,buffer0,1,block.size)
      BYTE.SLICE.INPUT(hostin1,buffer1,1,block.size)
      BYTE.SLICE.INPUT(hostin2,buffer2,1,block.size)
      BYTE.SLICE.INPUT(hostin3,buffer3,1,block.size)
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- all output and input in parallel
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
      hostin3 ? ch[3]
    TIME ? time0[0]
```

```
                PAR
                  BYTE.SLICE.OUTPUT(hostout0,buffer0,1,block.size)
                  BYTE.SLICE.OUTPUT(hostout1,buffer1,1,block.size)
                  BYTE.SLICE.OUTPUT(hostout2,buffer2,1,block.size)
                  BYTE.SLICE.OUTPUT(hostout3,buffer3,1,block.size)
                  BYTE.SLICE.INPUT(hostin0,buffer0,1,block.size)
                  BYTE.SLICE.INPUT(hostin1,buffer1,1,block.size)
                  BYTE.SLICE.INPUT(hostin2,buffer2,1,block.size)
                  BYTE.SLICE.INPUT(hostin3,buffer3,1,block.size)
                TIME ? time1[0]
                transfer.rate(time0[0],time1[0],1,block.size,rate)
                actual.rate := ((actual.rate * (j-1)) + rate)/j
            SKIP
            write.number (actual.rate)
            new.line
        SKIP
        new.line
        -- send to screen operations done concurrently
        IF
          cpumode = '0'
            write.string (" No other process running concurrently ")
          (((cpumode='2')OR(cpumode='4'))OR
                                  ((cpumode='6')OR(cpumode='8')))
            SEQ
              flag ! 'a'
              counter ? number
              write.string ("Number of operations (in //) at the ")
              write.string ("b001 transputer ")
              write.number (number)
              new.line
              hostin0 ? number
              write.string ("Number of operations (in //) at  ")
              write.string ("transputer 0 (b003) ")
              write.number (number)
          TRUE
            SEQ
              hostin0 ? number
              write.string ("Number of operations (in //) at ")
              write.string ("transputer 0 (b003) ")
              write.number (number)
        new.line
        new.line    :

SEQ  --- main byte.slice.transfer
  -- initializing buffers
  SEQ k = [1 FOR maxblock.size]
    SEQ
      buffer0 [BYTE k] := '0'
      buffer1 [BYTE k] := '1'
      buffer2 [BYTE k] := '2'
      buffer3 [BYTE k] := '3'
  SKIP
  IF
    cpumode = '2'
      PAR
        transfer (repetition, cpumode, flag, counter)
        cpubusysum (flag, counter)
    cpumode = '4'
      PRI PAR
        transfer (repetition, cpumode, flag, counter)
        cpubusysum (flag, counter)
    cpumode = '6'
      PAR
        transfer (repetition, cpumode, flag, counter)
        cpubusyprod (flag, counter)
    cpumode = '8'
      PRI PAR
        transfer (repetition, cpumode, flag, counter)
```

```
      cpubusyprod (flag, counter)
TRUE
   transfer (repetition, cpumode, flag, counter):
```

```
-- PROC int.transfer (VALUE repetition,cpumode)
PROC int.transfer ( VALUE repetition,cpumode)=
-- description
--- ***************************************************************
--- It initializes the buffers and it executes the procedure
--- intransfer,  and, when applicable one of the following:
---      cpubusy.prod or cpubusy.sum. (according to cpumode)
--- Uses global constant maxblock.size.
--- ***************************************************************

  -- variable declarations
  CHAN flag,    --- flags the cpu to stop
       counter: --- return the number of operations cpu did
  VAR wbuffer0 [maxwordblock.size + 1],
      wbuffer1 [maxwordblock.size + 1],
      wbuffer2 [maxwordblock.size + 1],
      wbuffer3 [maxwordblock.size + 1]:

  -- PROC intransfer (VALUE repetition, cpumode, CHAN flag, counter)
  PROC intransfer (VALUE repetition,cpumode,CHAN flag, counter)=
  -- description
  ---  ***************************************************************
  ---  Executes sequentially several parallel transfers of integers
  ---  to/from one to four transputers using input/output primitives
  ---  and output to the screen the transfer rate values of the
  ---  output TABLE.
  ---  Uses global constants : sizetable, nr.of.sizes, repetition
  ---  ***************************************************************

    -- variable declarations
    VAR block.size,
        actual.rate,
        rate,
        number,       --- the number of operations cpu did
        ch[4],
        deadtime, deadtime0, deadtime1, ----  to calculate deadtime
        time0[4],
        time1[4]:

    SEQ
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          -- making the table
          block.size := sizetable[i]
          write.number (block.size)
          Screen ! tab
          IF
            block.size < 4
              write.string("minimum transfer for integers ")
              write.string("is 4 bytes(word)")
            TRUE
              SEQ
                -- calculation of deadtime
                TIME ? deadtime0
                SEQ i = [1 FOR (block.size/4)]
                  SKIP
                TIME ? deadtime1
                deadtime := deadtime1 - deadtime0
                -- io handling
                -- output to one channel
                actual.rate := 0
                SEQ j = [1 FOR repetition]
                  SEQ
                    hostin0 ? ch[0]
                    TIME ? time0[0]
                    SEQ k = [1 FOR (block.size/4)]
                      hostout0 ! wbuffer0[k]
```

```
      TIME ? time1[0]
      time1[0] := time1[0] - deadtime
      transfer.rate (time0-0-, time1-0-, 1,
                              block.size, rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output/input from one channel
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    hostin0 ? ch[0]
    TIME ? time0[0]
    SEQ k = [1 FOR (block.size/4)]
      PAR
        hostout0 ! wbuffer0 [k]
        hostin0 ? wbuffer1 [k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output to two channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
    TIME ? time0[0]
    SEQ k = [1 FOR (block.size/4)]
      PAR
        hostout0 ! wbuffer0 [k]
        hostout1 ! wbuffer1 [k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output/input from two channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
    TIME ? time0[0]
    SEQ k = [ 1 FOR (block.size/4) ]
      PAR
        hostout0 ! wbuffer0 [k]
        hostout1 ! wbuffer1 [k]
        hostin0 ? wbuffer2 [k]
        hostin1 ? wbuffer3 [k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output to three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
```

```
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
    TIME ? time0[0]
    -- output handling
    SEQ k = [1 FOR (block.size/4)]
      PAR
        hostout0 ! wbuffer0 [k]
        hostout1 ! wbuffer1 [k]
        hostout2 ! wbuffer2 [k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- output/input from three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
    TIME ? time0[0]
    -- output/input handling
    SEQ k = [ 1 FOR (block.size/4) ]
      PAR
        hostout0 ! wbuffer0 [k]
        hostout1 ! wbuffer1 [k]
        hostout2 ! wbuffer2 [k]
        hostin0 ? wbuffer0 [k]
        hostin1 ? wbuffer1 [k]
        hostin2 ? wbuffer2 [k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- output to four channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
      hostin3 ? ch[3]
    TIME ? time0[0]
    -- input and output handling
    SEQ k = [1 FOR (block.size/4)]
      PAR
        hostout0 ! wbuffer0 [k]
        hostout1 ! wbuffer1 [k]
        hostout2 ! wbuffer2 [k]
        hostout3 ! wbuffer3 [k]
    TIME ? time1[0]
    time1[0] := time1[0] - deadtime
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
```

```
                Screen ! tab
                -- input from four channels
                actual.rate := 0
                SEQ j = [1 FOR repetition]
                  SEQ
                    PAR
                      hostin0 ? ch[0]
                      hostin1 ? ch[1]
                      hostin2 ? ch[2]
                      hostin3 ? ch[3]
                    TIME ? time0[0]
                    -- input handling
                    SEQ k = [ 1 FOR (block.size/4) ]
                      PAR
                        hostin0 ? wbuffer0 [k]
                        hostin1 ? wbuffer1 [k]
                        hostin2 ? wbuffer2 [k]
                        hostin3 ? wbuffer3 [k]
                    TIME ? time1[0]
                    time1[0] := time1[0] - deadtime
                    transfer.rate(time0[0],time1[0],1,block.size,rate)
                    actual.rate := ((actual.rate * (j-1)) + rate)/j
                SKIP
                write.number (actual.rate)
                Screen ! tab
                -- all output and input in parallel
                actual.rate := 0
                SEQ j = [1 FOR repetition]
                  SEQ
                    PAR
                      hostin0 ? ch[0]
                      hostin1 ? ch[1]
                      hostin2 ? ch[2]
                      hostin3 ? ch[3]
                    TIME ? time0[0]
                    -- input and output handling
                    SEQ k = [1 FOR (block.size/4)]
                      PAR
                        hostout0 ! wbuffer0 [k]
                        hostout1 ! wbuffer1 [k]
                        hostout2 ! wbuffer2 [k]
                        hostout3 ! wbuffer3 [k]
                        hostin0 ? wbuffer0 [k]
                        hostin1 ? wbuffer1 [k]
                        hostin2 ? wbuffer2 [k]
                        hostin3 ? wbuffer3 [k]
                    TIME ? time1[0]
                    time1[0] := time1[0] - deadtime
                    transfer.rate(time0[0],time1[0],1,block.size,rate)
                    actual.rate := ((actual.rate * (j-1)) + rate)/j
                SKIP
                write.number (actual.rate)
        new.line
    SKIP
    new.line
    -- send to screen operations done concurrently
    IF
      cpumode = '0'
        write.string (" No other process running concurrently ")
      (((cpumode='2')OR(cpumode='4'))OR
                                        ((cpumode='6')OR(cpumode='8')))
        SEQ
          flag ! 'a'
          counter ? number
          write.string ("Number of operations (in //) at the ")
          write.string ("b001 transputer ")
          write.number (number)
```

```
            new.line
            hostin0 ? number
            write.string ("Number of operations (in //) at  ")
            write.string ("transputer 0 (b003) ")
            write.number (number)
        TRUE
          SEQ
            hostin0 ? number
            write.string ("Number of operations (in //) at ")
            write.string ("transputer 0 (b003) ")
            write.number (number)
      new.line
      new.line    :

SEQ  --- main int.transfer
  -- buffers  initialization
  SEQ k = [1 FOR maxwordblock.size]
    SEQ
      wbuffer0 [k] := 10000
      wbuffer1 [k] := 20000
      wbuffer2 [k] := 30000
      wbuffer3 [k] := 40000
  SKIP
  IF
    cpumode = '2'
      PAR
        intransfer (repetition, cpumode, flag, counter)
        cpubusysum (flag, counter)
    cpumode = '4'
      PRI PAR
        intransfer (repetition, cpumode, flag, counter)
        cpubusysum (flag, counter)
    cpumode = '6'
      PAR
        intransfer (repetition, cpumode, flag, counter)
        cpubusyprod (flag, counter)
    cpumode = '8'
      PRI PAR
        intransfer (repetition, cpumode, flag, counter)
        cpubusyprod (flag, counter)
    TRUE
      intransfer (repetition, cpumode, flag, counter):
```

```
-- PROC word.slice.transfer (VALUE repetition, cpumode)
PROC word.slice.transfer (VALUE repetition, cpumode)=
-- description
--- ****************************************************************
--- It initializes the buffers and it executes the procedure
--- wordtransfer,  and, when applicable one of the following:
---     cpubusy.prod or cpubusy.sum. (according to cpumode)
--- Uses global constant maxblock.size.
--- ****************************************************************

  -- variable declarations
  CHAN flag,    --- flags the cpu to stop
       counter: --- return the number of operations cpu did
  VAR wbuffer0 [maxwordblock.size + 1],
      wbuffer1 [maxwordblock.size + 1],
      wbuffer2 [maxwordblock.size + 1],
      wbuffer3 [maxwordblock.size + 1]:

  -- PROC wordtransfer (VALUE repetition, cpumode, CHAN flag, ...)
  PROC wordtransfer (VALUE repetition,cpumode,CHAN flag, counter)=
  -- description
  ---  ************************************************************
  ---  Executes sequentially several parallel transfers of integers
  ---  to/from one to four transputers using the WORD.SLICE Procedure
  ---  and output to the screen the transfer rate values of the
  ---  output TABLE.
  ---  Uses global constants : sizetable, nr.of.sizes, repetition
  ---  ************************************************************

    -- variable declarations
    VAR block.size,      --- number of bytes to be transmited
        actual.rate,      --- updated rate and final rate
        rate,           --- auxiliary variable to hold temporary rate
        number,           --- the number of operations cpu did
        ch[4],
        time0[4],
        time1[4]:

    SEQ
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          -- making the table after each io operation
          block.size := sizetable[i]
          write.number (block.size)
          Screen ! tab
          IF
            block.size < 4
              write.string("minimum transfer for integers ")
              write.string("is 4 bytes(word)")
            TRUE


--- ****************************************************************
--- ATTENTION ! The code is shifted left 12 spaces from here on,
---             due to printing requirements.
--- ****************************************************************
    SEQ
      -- output to one channel
      actual.rate := 0
      SEQ j = [1 FOR repetition]
        SEQ
          hostin0 ? ch[0]
          TIME ? time0[0]
          WORD.SLICE.OUTPUT(hostout0,wbuffer0,1,(block.size/4))
          TIME ? time1[0]
```

```
      transfer.rate (time0[0],time1[0],1,block.size,rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output/input in one link
actual.rate := 0
SEQ j = [1 FOR repetition]
   SEQ
      hostin0 ? ch[0]
      TIME ? time0[0]
      PAR
         WORD.SLICE.OUTPUT(hostout0,wbuffer0,1,(block.size/4))
         WORD.SLICE.INPUT(hostin0,wbuffer1,1,(block.size/4))
      TIME ? time1[0]
      transfer.rate(time0[0],time1[0],1,block.size,rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output to two channels
actual.rate := 0
SEQ j = [1 FOR repetition]
   SEQ
      PAR
         hostin0 ? ch[0]
         hostin1 ? ch[1]
      TIME ? time0[0]
      PAR
         WORD.SLICE.OUTPUT(hostout0,wbuffer0,1,(block.size/4))
         WORD.SLICE.OUTPUT(hostout1,wbuffer1,1,(block.size/4))
      TIME ? time1[0]
      transfer.rate(time0[0],time1[0],1,block.size,rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output/input in two links
actual.rate := 0
SEQ j = [1 FOR repetition]
   SEQ
      PAR
         hostin0 ? ch[0]
         hostin1 ? ch[1]
      TIME ? time0[0]
      PAR
         WORD.SLICE.OUTPUT(hostout0,wbuffer0,1,(block.size/4))
         WORD.SLICE.OUTPUT(hostout1,wbuffer1,1,(block.size/4))
         WORD.SLICE.INPUT(hostin0,wbuffer2,1,(block.size/4))
         WORD.SLICE.INPUT(hostin1,wbuffer3,1,(block.size/4))
      TIME ? time1[0]
      transfer.rate(time0[0],time1[0],1,block.size,rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab

-- output to three channels
actual.rate := 0
SEQ j = [1 FOR repetition]
   SEQ
      PAR
         hostin0 ? ch[0]
         hostin1 ? ch[1]
         hostin2 ? ch[2]
      TIME ? time0[0]
      PAR
         WORD.SLICE.OUTPUT(hostout0,wbuffer0,1,(block.size/4))
         WORD.SLICE.OUTPUT(hostout1,wbuffer1,1,(block.size/4))
```

```
        WORD.SLICE.OUTPUT(hostout2,wbuffer2,1,(block.size/4))
      TIME ? time1[0]
      transfer.rate(time0[0],time1[0],1,block.size,rate)
      actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- output/input in three links
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
    TIME ? time0[0]
    PAR
      WORD.SLICE.OUTPUT(hostout0,wbuffer0,1,(block.size/4))
      WORD.SLICE.OUTPUT(hostout1,wbuffer1,1,(block.size/4))
      WORD.SLICE.OUTPUT(hostout2,wbuffer2,1,(block.size/4))
      WORD.SLICE.INPUT(hostin0,wbuffer0,1,(block.size/4))
      WORD.SLICE.INPUT(hostin1,wbuffer1,1,(block.size/4))
      WORD.SLICE.INPUT(hostin2,wbuffer2,1,(block.size/4))
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- output to four channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
      hostin3 ? ch[3]
    TIME ? time0[0]
    PAR
      WORD.SLICE.OUTPUT(hostout0,wbuffer0,1,(block.size/4))
      WORD.SLICE.OUTPUT(hostout1,wbuffer1,1,(block.size/4))
      WORD.SLICE.OUTPUT(hostout2,wbuffer2,1,(block.size/4))
      WORD.SLICE.OUTPUT(hostout3,wbuffer3,1,(block.size/4))
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
SKIP
write.number (actual.rate)
Screen ! tab
-- input from four channels
actual.rate := 0
SEQ j = [1 FOR repetition]
  SEQ
    PAR
      hostin0 ? ch[0]
      hostin1 ? ch[1]
      hostin2 ? ch[2]
      hostin3 ? ch[3]
    TIME ? time0[0]
    PAR
      WORD.SLICE.INPUT(hostin0,wbuffer0,1,(block.size/4))
      WORD.SLICE.INPUT(hostin1,wbuffer1,1,(block.size/4))
      WORD.SLICE.INPUT(hostin2,wbuffer2,1,(block.size/4))
      WORD.SLICE.INPUT(hostin3,wbuffer3,1,(block.size/4))
    TIME ? time1[0]
    transfer.rate(time0[0],time1[0],1,block.size,rate)
    actual.rate := ((actual.rate * (j-1)) + rate)/j
```

```
      SKIP
      write.number (actual.rate)
      Screen ! tab
      -- all output and input in parallel
      actual.rate := 0
      SEQ j = [1 FOR repetition]
        SEQ
          PAR
            hostin0 ? ch[0]
            hostin1 ? ch[1]
            hostin2 ? ch[2]
            hostin3 ? ch[3]
          TIME ? time0[0]
          PAR
            WORD.SLICE.OUTPUT(hostout0,wbuffer0,1,(block.size/4))
            WORD.SLICE.OUTPUT(hostout1,wbuffer1,1,(block.size/4))
            WORD.SLICE.OUTPUT(hostout2,wbuffer2,1,(block.size/4))
            WORD.SLICE.OUTPUT(hostout3,wbuffer3,1,(block.size/4))
            WORD.SLICE.INPUT(hostin0,wbuffer0,1,(block.size/4))
            WORD.SLICE.INPUT(hostin1,wbuffer1,1,(block.size/4))
            WORD.SLICE.INPUT(hostin2,wbuffer2,1,(block.size/4))
            WORD.SLICE.INPUT(hostin3,wbuffer3,1,(block.size/4))
          TIME ? time1[0]
          transfer.rate(time0[0],time1[0],1,block.size,rate)
          actual.rate := ((actual.rate * (j-1)) + rate)/j
      SKIP
--- ***************************************************************
--- ATTENTION ! End of code shifted 12 spaces to the left.
--- ***************************************************************



                    write.number (actual.rate)
          new.line
        SKIP
        new.line
        -- send to screen operations done concurrently
        IF
          cpumode = '0'
            write.string (" No other process running concurrently ")
          (((cpumode='2')OR(cpumode='4'))
                                      OR((cpumode='6')OR(cpumode='8')))
            SEQ
              flag ! 'a'
              counter ? number
              write.string ("Number of operations (in //) at the ")
              write.string ("b001 transputer ")
              write.number (number)
              new.line
              hostin0 ? number
              write.string ("Number of operations (in //) at  ")
              write.string ("transputer 0 (b003) ")
              write.number (number)
          TRUE
            SEQ
              hostin0 ? number
              write.string ("Number of operations (in //) at ")
              write.string ("transputer 0 (b003) ")
              write.number (number)
        new.line
        new.line    :



    SEQ   --- main word.slice.transfer
      -- word buffers  initialization
      SEQ k = [1 FOR maxwordblock.size]
```

```
  SEQ
    wbuffer0 [k] := 10000
    wbuffer1 [k] := 20000
    wbuffer2 [k] := 30000
    wbuffer3 [k] := 40000
SKIP
IF
  cpumode = '2'
    PAR
      wordtransfer (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  cpumode = '4'
    PRI PAR
      wordtransfer (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  cpumode = '6'
    PAR
      wordtransfer (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  cpumode = '8'
    PRI PAR
      wordtransfer (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    wordtransfer (repetition, cpumode, flag, counter):
```

```
-- SC PROC get.choices (CHAN Keyboard, Screen, VAR ch, ..., run)
-- PROC get.choices (CHAN Keyboard, Screen, VAR ch, cpumode, run)
PROC get.choices (CHAN Keyboard, Screen, VAR ch, cpumode, run)=
---  ************************************************************
---  presents menus and gets user's choices of cpumode and construct
---  ************************************************************
  -- modlibrary.occ
  -- io_routines.occ  (partial)
  -- SC PROC new.line (CHAN Screen)
  -- PROC new.line (CHAN Screen)
  PROC new.line (CHAN Screen)=
  ---  ************************************************************
  ---  jumps a line on the screen. May be compiled separately
  ---  ************************************************************
    DEF EndBuffer = -3:
    DEF cr = 13 :
    DEF lf = 10 :
    SEQ
      Screen ! cr;lf;EndBuffer :
  -- descriptor
  -- code
  -- SC PROC write (CHAN Screen, VALUE string[])
  -- PROC write (CHAN Screen, VALUE string[])
  PROC write (CHAN Screen, VALUE string[]) =
  --- ************************************************************
  --- Writes a given string to the screen, in a byte by byte fashion
  --- May be compiled separately
  --- ************************************************************
    DEF EndBuffer = -3:
    SEQ
      SEQ i = [1 FOR string[BYTE 0]]
        Screen ! string[BYTE i]
      Screen ! EndBuffer :
  -- descriptor
  -- code
  -- SC PROC clear (CHAN Screen)
  -- PROC clear (CHAN Screen)
  PROC clear (CHAN Screen)=
  --- ************************************************************
  --- clears the screen. May be compiled separately
  --- ************************************************************
    DEF EndBuffer = -3:
    DEF esc = 27:
    SEQ
      Screen ! esc; '-'; '2'; 'J'; EndBuffer --- clear sequence
      Screen ! esc; '['; 'H' :               --- home cursor
  -- descriptor
  -- code
  -- SC PROC write.number (CHAN Screen, VALUE number)
  -- PROC write.number (CHAN Screen, VALUE number)
  PROC write.number(CHAN Screen, VALUE number) =
  --- ************************************************************
  --- This PROC outputs a signed integer value to the screen
  --- May be compiled separately
  --- ************************************************************
    VAR output[16], count, x:
    SEQ
      x:= number
      count:= 0
      IF
        --  handle special cases
```

```
        x=0
          Screen ! '0'
        x<0
          SEQ
            Screen ! '-'
            x:=-x
        TRUE
          SKIP
      WHILE x>0
        --  construct number
        SEQ
          output[count] := (x   10) + '0'
          count := count + 1
          x:= x/10
      WHILE count > 0
        --  output number
        SEQ
          count := count-1
          Screen ! output[count]
      SKIP:
-- descriptor
-- code
-- utilities.occ  (partial)
-- SC PROC capitalize (VAR ch)
-- PROC capitalize (VAR ch)
PROC capitalize (VAR ch) =
---  ************************************************************
--- capitalizes any lower case character into upper case
---  ************************************************************
  DEF delta =('a' - 'A') :
                                  ---  A ---> 65
                                  ---  a ---> 97      ASCII values
                                  ---  z ---> 122
  SEQ
    IF
      (ch <= 'z') AND (ch >= 'a')
        ch := ch - delta
      TRUE
        SKIP :
-- descriptor
-- code
-- global_def.tds (partial)
-- Constants Definitions
DEF EndBuffer = -3:
DEF tab       =  9:
DEF lf        = 10:
DEF cr        = 13:
DEF esc       = 27:
DEF sp        = 32:

-- PROC write.header
PROC write.header =
---  ************************************************************
---  writes the header of the output table
---  ************************************************************
  SEQ
    run := run + 1
    clear(Screen)
    write(Screen, "RUN # ")
    write.number (Screen,run)
    Screen ! sp;sp;sp
    -- output the cpu mode to the screen
    IF
```

```
        cpumode = '0'
          write(Screen, "cpu mode = 0 (no par proc )   ")
        cpumode = '1'
          write(Screen, "cpu mode = 1 (one sum par )   ")
        cpumode = '2'
          write(Screen, "cpu mode = 2 (all sum par )   ")
        cpumode = '3'
          write(Screen, "cpu mode = 3 (one sum pripar) ")
        cpumode = '4'
          write(Screen, "cpu mode = 4 (all sum pripar) ")
        cpumode = '5'
          write(Screen, "cpu mode = 5 (one prod par)   ")
        cpumode = '6'
          write(Screen, "cpu mode = 6 (all prod par)   ")
        cpumode = '7'
          write(Screen, "cpu mode = 7 (one prod pripar)")
        cpumode = '8'
          write(Screen, "cpu mode = 8 (all prod pripar)")
        TRUE
          SKIP
      Screen ! sp;sp
      -- output the construct type to the screen
      IF
        ch = 'A'
          write(Screen, "input/output channels    (bytes) ")
        ch = 'B'
          write(Screen, "BYTE.SLICE.input/output  (bytes) ")
        ch = 'I'
          write(Screen, "input/output channels  (integers)")
        ch = 'W'
          write(Screen, "WORD.SLICE.input/output(integers)")
        TRUE
          SKIP
      new.line (Screen)
      new.line (Screen)
      write(Screen, "BYTES  1 OUT   1IN/OUT  2 OUT  2IN/OUT  3 OUT")
      write(Screen, "3IN/OUT  4 OUT   4 IN   4IN/OUT")
      new.line (Screen):
VAR answer :
SEQ
  IF
    run = 0
      SEQ
        -- output to the screen presentation of program
        clear(Screen)
        write(Screen, "     This is an Evaluation Program for ")
        write(Screen, "the Transputer")
        new.line (Screen)
        write(Screen, " It is fully interactive and you will ")
        write(Screen, "be prompted in")
        new.line (Screen)
        write(Screen, " each run to choose cpu mode and type ")
        write(Screen, "of construct ")
        new.line (Screen)
        write(Screen, " The output table will present transfer ")
        write(Screen, " rates in ")
        new.line (Screen)
        write(Screen, " Kbits/sec for the 16 different ")
        write(Screen, "block.sizes and the 9")
        new.line (Screen)
        write(Screen, " channel configurations ")
        new.line (Screen)
        new.line (Screen)
        write(Screen, " TYPE (Y)ES  if you want to use it ")
        new.line (Screen)
        write(Screen, "      (N)O   if you want to quit ")
        new.line (Screen)
        answer := 'z'
```

```
            -- validate answer
            WHILE ((answer <> 'Y') AND (answer <> 'N'))
              SEQ
                write(Screen, " Type your choice ")
                Keyboard ? answer
                capitalize (answer)
                Screen   ! answer
                new.line (Screen)
        TRUE
          SKIP
      clear(Screen)
      -- choosing type of construct
      -- prompt for type of construct
      write(Screen, " Choose type of construct to be used ")
      new.line(Screen)
      write(Screen, "  A  for input/output channels   (bytes) ")
      new.line(Screen)
      write(Screen, "  B  for BYTE.SLICE input/output (bytes)")
      new.line(Screen)
      write(Screen, "  I  for input/output channels   (words) ")
      new.line(Screen)
      write(Screen, "  W  for WORD.SLICE input/output (words)")
      new.line(Screen)
      -- validate type of construct
      ch := 'Z'
      WHILE (((ch <> 'A')AND(ch <> 'B'))AND((ch <> 'W')AND(ch <> 'I')))
        SEQ
          write(Screen, " Type your choice ")
          Keyboard ? ch
          capitalize (ch)
          Screen ! ch
          new.line(Screen)
      new.line(Screen)
      -- choosing cpumode during transfers
      -- prompt for cpu mode
      write(Screen, " Choose cpu mode during transfers")
      new.line(Screen)
      write(Screen, "0  -> cpus executing no concurrent processes ")
      new.line(Screen)
      write(Screen, "1  -> B003 cpus executing sum concurrently (par)")
      new.line(Screen)
      write(Screen, "2  -> all cpus executing sum concurrently (par)")
      new.line(Screen)
      write(Screen, "3  -> B003 cpus executing sum concurrently(pripar)")
      new.line(Screen)
      write(Screen, "4  -> all cpus executing sum concurrently (pripar)")
      new.line(Screen)
      write(Screen, "5  -> B003 cpus executing array products   (par)")
      new.line(Screen)
      write(Screen, "6  -> all cpus executing array products  (par)")
      new.line(Screen)
      write(Screen, "7  -> B003 cpus executing array products (pripar)")
      new.line(Screen)
      write(Screen, "8  -> all cpus executing array products  (pripar)")
      new.line(Screen)
      -- validate cpu mode
      cpumode  := 10
      WHILE  ((cpumode > #38) OR (cpumode < #30 ))
                              --- 0 < cpumode < 8 (IN ASCII)
        SEQ
          write(Screen, " Type your choice ")
          Keyboard ? cpumode
          Screen ! cpumode
          new.line(Screen)
      write.header:
-- descriptor
```

```
-- code
-- PROC user.interface
PROC user.interface =
---  ************************************************************
---  Presents menus and calls right modules to be executed
---  in the transputer root.
---  ************************************************************

  -- constant and variable declarations
  VAR  run        : ---  number of runs made this time (RUN #)
  VAR  answer     : ---  users choice in continue or quit
  VAR  construct  : ---  users choice of construct
  VAR  cpumode    : ---  users choice of cpu mode while transfering
                     ---  data


  SEQ
    run := 0
    answer := 'z'
    clear.screen
    write.string (" Do you want to use the Link Evaluation Program?")
    -- validate answer
    WHILE ((answer <> 'Y') AND (answer <> 'N'))
      SEQ
        new.line
        write.string (" Type your choice (Y) or (N)")
        Keyboard ? answer
        capitalize (answer)
        Screen   ! answer
        new.line

    WHILE  answer = 'Y'
      SEQ
        get.choices (Keyboard, Screen, construct, cpumode, run)
        -- send choices to other transputers
        PAR
          hostout0 ! construct; cpumode; repetition
          hostout1 ! construct; cpumode; repetition
          hostout2 ! construct; cpumode; repetition
          hostout3 ! construct; cpumode; repetition

        -- executing the right procedure and prompting for new run
        IF
          construct = 'A'
            inout.transfer (repetition, cpumode)
          construct = 'B'
            byte.slice.transfer (repetition, cpumode)
          construct = 'I'
            int.transfer (repetition, cpumode)
          construct = 'W'
            word.slice.transfer (repetition, cpumode)
          TRUE
            SKIP
        -- prompt for another run and validate answer
        answer := 'Z'    --- to make the next loop be executed
        WHILE ((answer <> 'Y') AND (answer <> 'N'))
          SEQ
            write.string("Do you want another run? Type (Y) or (N)")
            Keyboard ? answer
            capitalize (answer)
            Screen   ! answer
            new.line

        -- send answer to other transputers
        PAR
          hostout0 ! answer
          hostout1 ! answer
          hostout2 ! answer
          hostout3 ! answer

    clear.screen
    write.string (" Thank you for using the Link Evaluation Program")
```

```
        new.line
        write.string (" Press reset on the b001 board to get back ")
        write.string (" to VAX/VMS ") :

    PAR
      IMS.B001.terminal.driver(Keyboard,Screen,port,baud)
      user.interface:
--- ***************************************************************
--- END OF CODE IN TRANSPUTER ROOT
--- ***************************************************************
```

```
-- TRANSPUTER0_B003.TDS
-- SC PROC transfer0.b003
-- PROC transfer0.b003 (CHAN in,out)
PROC transfer0.b003 (CHAN in,out) =
-- description
--- ****************************************************************
--- This is the outer procedure placed on transputer 0 . It contains
--- global variables and constants, and all procedures that run in this
--- transputer. It receives a construct type (ch), cpu mode (cpumode),
--- and number of times each communication sequence (repetition), and
--- calls accordingly one of the following procedures:
---         - io.transfer0,
---         - byte.slice.transfer0,
---         - int.transfer0 or
---         - word.slice.transfer0
--- ****************************************************************

  -- Link Definitions
  DEF link0in  = 4 :
  DEF link0out = 0 :
  DEF link1in  = 5 :
  DEF link1out = 1 :
  DEF link2in  = 6 :
  DEF link2out = 2 :
  DEF link3in  = 7 :
  DEF link3out = 3 :

  -- constant declarations
  DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                          1024, 1280, 2048, 4096, 8192, 10000 ]:
  DEF nr.of.sizes = 16:          --- as counted from above table
  DEF maxblock.size = 10000:     --- last from the above table
  DEF maxwordblock.size = maxblock.size / 4:

  -- variables declarations
  VAR ch         :   --- choice of the user in type of construct
  VAR answer     :   --- choice of the user in continue
  VAR cpumode    :   --- choice of the user in cpu operation concurrently
  VAR repetition:    --- choice of the user in number of times to run

  -- SC PROC cpubusysum (CHAN flag1, counterchan)
  -- PROC cpubusysum (CHAN flag1,counterchan)
  PROC cpubusysum (CHAN flag1,counterchan)=
  -- description
  ---  ****************************************************************
  ---  It keeps the cpu working in parallel (time sharing) with link
  ---  transfers by doing sum operations. It Stops when receives
  ---  a flag by the channel flag1 from the procedure transfer that
  ---  is being executed concurrently.
  ---  Outputs by channel counterchan  number of operations done.
  ---  ****************************************************************

    VAR a,b,e,
        working,
        counter,
        ch :

    SEQ
      counter := 0
      working := TRUE
      TIME ? a
      WHILE working
        ALT
          flag1 ? ch
            working := FALSE
          TIME ? b
            SEQ
              e := a + b
              counter := counter + 1
      counterchan ! counter:

  -- descriptor
```

```
-- code
-- SC PROC cpubusyprod (CHAN flag1,counterchan)
-- PROC cpubusyprod (CHAN flag1,counterchan)
PROC cpubusyprod (CHAN flag1,counterchan)=
-- description
---  ***************************************************************
---  It keeps the cpu working in parallel(time sharing) with the link
---  transfers by doing array multiplications. It stops when receives
---  a flag by the channel flag1 from the transfer procedure, that is
---  being executed concurrently. It outputs by channel counterchan
---  the number of operations done.
---  ***************************************************************

  -- constants and variable declarations
  DEF number = 100:   ---- size of array
  VAR a[number + 1],  ---- array of integers
      b[number + 1],  ---- array of integers
      e[number + 1],  ---- array of integers
      clock,          ---- integer -variable to get time
      working,        ---- boolean -to stop execution
      counter,        ---- integer -number of operations done
      ch :

  SEQ
    -- initialize buffers and variables
    SEQ i = [ 1 FOR number ]
      SEQ
        a[i] := 3*i
        b[i] := 5*i
    SKIP
    counter := 0
    working := TRUE
    WHILE working
      ALT
        flag1 ? ch
          working := FALSE
        TIME ? clock
          SEQ
            SEQ i = [1 FOR number]
              e[i] := a[i] * b[i]
            counter := counter + number ---updates nr. of operations
    counterchan ! counter:
```

```
-- PROC inout.transfer0 (VALUE repetition,cpumode)
PROC inout.transfer0 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures iotransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC iotransfer0 (VALUE repetition, cpumode, CHAN flag, counter)
  PROC iotransfer0 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description.io
  --- ************************************************************
  --- It executes sequentially several parallel transfers using the
  --- input/output primitives to/from transputer root.
  --- It uses the global constants sizetable, nr.of.sizes, repetition
  --- ************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR buffer0 [BYTE maxblock.size + 1]:
    VAR buffer1 [BYTE maxblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxblock.size]
        SEQ
          buffer0-BYTE i- := i\8
          buffer1-BYTE i- := i\8
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          block.size := sizetable[i]
          -- input and output handling
          -- input from one channel
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                in ? buffer0[BYTE k]
          SKIP
          -- input/output to/from one link
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                PAR
                  in ? buffer0[BYTE k]
                  out ! buffer1[BYTE k]
          SKIP
          -- input from two channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                in ? buffer0[BYTE k]
          SKIP
          -- input/output to two links
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
```

```
            SEQ k = [1 FOR block.size]
              PAR
                in ? buffer0[BYTE k]
                out ! buffer1[BYTE k]
        SKIP
        -- input from three channels
        SEQ j = [1 FOR repetition]
          SEQ
            out ! 'a'
            SEQ k = [1 FOR block.size]
              in ? buffer0[BYTE k]
        SKIP
        -- input/output to three links
        SEQ j = [1 FOR repetition]
          SEQ
            out ! 'a'
            SEQ k = [1 FOR block.size]
              PAR
                out ! buffer0[BYTE k]
                in ? buffer1[BYTE k]
        SKIP
        -- input from four channels
        SEQ j = [1 FOR repetition]
          SEQ
            out ! 'a'
            SEQ k = [1 FOR block.size]
              in ? buffer0[BYTE k]
        SKIP
        -- output to four channels
        SEQ j = [1 FOR repetition]
          SEQ
            out ! 'a'
            SEQ k = [1 FOR block.size]
              out ! buffer0[BYTE k]
        SKIP
        -- all output and input in parallel
        SEQ j = [1 FOR repetition]
          SEQ
            out ! 'a'
            SEQ k = [1 FOR block.size]
              PAR
                in ? buffer0[BYTE k]
                out ! buffer1[BYTE k]
        SKIP
    SKIP
    IF
      -- cpumode not = '0' then get the number of computations done
      cpumode <> '0'
        SEQ
          flag ! 'a'      --- flag to stop procedure cpubusy
          opnumber ? number  --- receiving computations from cpubusy
          out ! number   --- sending computations to transputer root
      TRUE
        SKIP  :

-- main PROC inout.transfer0
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      iotransfer0 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      iotransfer0 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
```

```
  PAR
    iotransfer0 (repetition, cpumode, flag, counter)
    cpubusyprod (flag, counter)
((cpumode = '7') OR (cpumode = '8'))
  PRI PAR
    iotransfer0 (repetition, cpumode, flag, counter)
    cpubusyprod (flag, counter)
TRUE
  iotransfer0 (repetition, cpumode, flag, counter):
```

```
-- PROC byte.slice.transfer0 (VALUE repetition,cpumode)
PROC byte.slice.transfer0 (VALUE repetition,cpumode)=
-- description.
--- ****************************************************************
--- Initializes the buffers and executes the procedures transfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC transfer0 (VALUE repetition, cpumode, CHAN flag, counter)
  PROC transfer0 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ****************************************************************
  --- It executes sequentially several parallel transfers using the
  --- BYTE.SLICE procedures  to/from transputer root.
  --- It uses the global constants sizetable, nr.of.sizes, repetition
  --- ****************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR buffer0 [BYTE maxblock.size + 1]:
    VAR buffer1 [BYTE maxblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxblock.size]
        SEQ
          buffer0-BYTE i- := i\8
          buffer1-BYTE i- := i\8
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          block.size := sizetable[i]
          -- input and output handling
          -- input from one channel
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- input/output to one channel
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
          -- input from two channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- input/output to two channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
```

```
          SKIP
          -- input from three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- input/output to three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
          -- input from four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- output to four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.OUTPUT(out,buffer0,1,block.size)
          SKIP
          -- all output and input in parallel
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
      SKIP
      IF
        -- cpumode not = '0' then get the number of computations done.
        cpumode <> '0'
          SEQ
            flag ! 'a'
            opnumber ? number
            out ! number
        TRUE
          SKIP  :

-- main PROC byte.slice.transfer0
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      transfer0 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      transfer0 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      transfer0 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      transfer0 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    transfer0 (repetition, cpumode, flag, counter):
```

```
-- PROC int.transfer0 (VALUE repetition,cpumode)
PROC int.transfer0 (VALUE repetition,cpumode)=
-- description.
--- ****************************************************************
--- Initializes the buffers and executes the procedures intransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxwordblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC intransfer0 (VALUE repetition, cpumode, CHAN flag, counter)
  PROC intransfer0 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ****************************************************************
  --- It executes sequentially several parallel transfers using the
  --- input/output primitives to/from transputer root.
  --- It uses the global constants sizetable, nr.of.sizes, repetition
  --- ****************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR wbuffer0 [maxwordblock.size + 1]:
    VAR wbuffer1 [maxwordblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxwordblock.size]
        SEQ
          wbuffer0[i] := i
          wbuffer1[i] := i
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        IF
          sizetable[i] < 4
            SKIP ---minimum number of bytes is 4 for integer transfer
          TRUE
            SEQ
              block.size := sizetable[i]
              -- input and output handling
              -- input from one channel
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    in ? wbuffer0[k]
              SKIP
              -- input/output to one link
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    PAR
                      in ? wbuffer0[k]
                      out ! wbuffer1[k]
              SKIP
              -- input from two channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    in ? wbuffer0[k]
              SKIP
```

```
            -- input/output to two links
            SEQ j = [1 FOR repetition]
              SEQ
                out ! 'a'
                SEQ k = [1 FOR (block.size/4)]
                  PAR
                    in ? wbuffer0[k]
                    out ! wbuffer1[k]
            SKIP
            -- input from three channels
            SEQ j = [1 FOR repetition]
              SEQ
                out ! 'a'
                SEQ k = [1 FOR (block.size/4)]
                  in ? wbuffer0[k]
            SKIP
            -- input/output to three links
            SEQ j = [1 FOR repetition]
              SEQ
                out ! 'a'
                SEQ k = [1 FOR (block.size/4)]
                  PAR
                    in ? wbuffer0[k]
                    out ! wbuffer1[k]
            SKIP
            -- input from four channels
            SEQ j = [1 FOR repetition]
              SEQ
                out ! 'a'
                SEQ k = [1 FOR (block.size/4)]
                  in ? wbuffer0[k]
            SKIP
            -- output to four channels
            SEQ j = [1 FOR repetition]
              SEQ
                out ! 'a'
                SEQ k = [1 FOR (block.size/4)]
                  out ! wbuffer0[k]
            SKIP
            -- all output and input in parallel
            SEQ j = [1 FOR repetition]
              SEQ
                out ! 'a'
                SEQ k = [1 FOR (block.size/4)]
                  PAR
                    in ? wbuffer0[k]
                    out ! wbuffer1[k]
            SKIP
      SKIP
      IF
        -- cpumode not = '0' then get the number of computations done.
        cpumode <> '0'
          SEQ
            flag ! 'a'
            opnumber ? number
            out ! number
        TRUE
          SKIP  :

-- main PROC int.transfer0
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      intransfer0 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
```

```
((cpumode = '3') OR (cpumode = '4'))
  PRI PAR
    intransfer0 (repetition, cpumode, flag, counter)
    cpubusysum (flag, counter)
((cpumode = '5') OR (cpumode = '6'))
  PAR
    intransfer0 (repetition, cpumode, flag, counter)
    cpubusyprod (flag, counter)
((cpumode = '7') OR (cpumode = '8'))
  PRI PAR
    intransfer0 (repetition, cpumode, flag, counter)
    cpubusyprod (flag, counter)
TRUE
  intransfer0 (repetition, cpumode, flag, counter):
```

```
-- PROC word.slice.transfer0 (VALUE repetition,cpumode)
PROC word.slice.transfer0 (VALUE repetition,cpumode)=
-- description.
--- ****************************************************************
--- Initializes the buffers and executes the procedures wordtransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxwordblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC wordtransfer0 (VALUE repetition, cpumode, CHAN flag,...)
  PROC wordtransfer0 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ************************************************************
  --- It executes sequentially several parallel transfers using the
  --- WORD.SLICE procedures  to/from transputer root.
  --- It uses the global constants sizetable, nr.of.sizes, repetition
  --- ************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR wbuffer0 [ maxwordblock.size + 1]:
    VAR wbuffer1 [ maxwordblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxwordblock.size]
        SEQ
          wbuffer0[i] := i
          wbuffer1[i] := i
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        IF
          sizetable[i] < 4
            SKIP ---minimum number of bytes is 4 for integer transfer
          TRUE
            SEQ
              block.size := sizetable[i]
              -- input and output handling
              -- input from one channel
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
              SKIP
              -- input/output to one link
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  PAR
                    WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                    WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
              SKIP
              -- input from two channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
              SKIP
              -- input/output to two links
              SEQ j = [1 FOR repetition]
                SEQ
```

```
                    out ! 'a'
                    PAR
                      WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                      WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
                SKIP
                -- input from three channels
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                SKIP
                -- input/output to three links
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    PAR
                      WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                      WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
                SKIP
                -- input from four channels
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                SKIP
                -- output to four channels
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    WORD.SLICE.OUTPUT(out,wbuffer0,1,(block.size/4))
                SKIP
                -- all output and input in parallel
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    PAR
                      WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                      WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
                SKIP
    SKIP
    IF
      -- cpumode not = '0' then get the number of computations done.
      cpumode <> '0'
        SEQ
          flag ! 'a'
          opnumber ? number
          out ! number
      TRUE
        SKIP  :

-- main PROC word.slice.transfer0
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      wordtransfer0 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      wordtransfer0 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      wordtransfer0 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
```

```
            wordtransfer0 (repetition, cpumode, flag, counter)
            cpubusyprod (flag, counter)
        TRUE
          wordtransfer0 (repetition, cpumode, flag, counter):
  -- procedure body transfer0.b003
  SEQ
    answer := 'Y'
    WHILE  answer = 'Y'
      SEQ
        in ? ch
        in ? cpumode
        in ? repetition
        IF
          ch = 'A'
            inout.transfer0 (repetition,cpumode)
          ch = 'B'
            byte.slice.transfer0 (repetition,cpumode)
          ch = 'I'
            int.transfer0 (repetition,cpumode)
          ch = 'W'
            word.slice.transfer0 (repetition,cpumode)
          TRUE
            SKIP
        in ? answer :
--- ************************************************************************
--- END OF CODE IN TRANSPUTER 0 B003
--- ************************************************************************
```

```
-- TRANSPUTER1_B003.TDS
-- SC PROC transfer1.b003
-- PROC transfer1.b003 (CHAN in,out)
PROC transfer1.b003 (CHAN in,out) =
-- description
--- ********************************************************************
--- This is the outer procedure placed on transputer 1 . It contains
--- global variables and constants, and all procedures that run in this
--- transputer. It receives a construct type (ch), cpu mode (cpumode),
--- and number of times each communication sequence (repetition), and
--- calls accordingly one of the following procedures:
---         - io.transfer1,
---         - byte.slice.transfer1,
---         - int.transfer1 or
---         - word.slice.transfer1
--- ********************************************************************

  -- Link Definitions
  DEF link0in  = 4 :
  DEF link0out = 0 :
  DEF link1in  = 5 :
  DEF link1out = 1 :
  DEF link2in  = 6 :
  DEF link2out = 2 :
  DEF link3in  = 7 :
  DEF link3out = 3 :
  -- constant declarations
  DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                          1024, 1280, 2048, 4096, 8192, 10000 ]:
  DEF nr.of.sizes = 16:        --- as counted from above table
  DEF maxblock.size = 10000:   --- last from the above table
  DEF maxwordblock.size = maxblock.size/4:
  -- variable declarations
  VAR ch        :   --- choice of the user in type of construct
  VAR answer    :   --- choice of the user in continue
  VAR cpumode   :   --- choice of the user in cpu operation concurrently
  VAR repetition:   --- choice of the user in number of times to run
  -- SC PROC cpubusysum (CHAN flag1, counterchan)
  -- PROC cpubusysum (CHAN flag1,counterchan)
  PROC cpubusysum (CHAN flag1,counterchan)=
  -- description
  ---  ********************************************************************
  ---  It keeps the cpu working in parallel (time sharing) with link
  ---  transfers by doing sum operations . It Stops when it receives
  ---  a flag by the channel flag1 from the transfer procedure that is
  ---  being executed concurrently. It Outputs by channel counterchan
  ---  the number of operations done.
  ---  ********************************************************************

    VAR a,b,e,
        working,
        counter,
        ch :
    SEQ
      counter := 0
      working := TRUE
      TIME ? a
      WHILE working
        ALT
          flag1 ? ch
            working := FALSE
          TIME ? b
            SEQ
              e := a + b
              counter := counter + 1
      counterchan ! counter:
```

```
-- descriptor
-- code
-- SC PROC cpubusyprod (CHAN flag1,counterchan)
-- PROC cpubusyprod (CHAN flag1,counterchan)
PROC cpubusyprod (CHAN flag1,counterchan)=
-- description
---  ****************************************************************
---  It keeps the cpu working in parallel(time sharing) with the link
---  transfers by doing array multiplications. It stops when receives
---  a flag by the channel flag1 from the transfer procedure, that is
---  being executed concurrently. It outputs by channel counterchan
---  the number of operations done.
---  ****************************************************************
  -- constants and variable declarations
  DEF number = 100:    ---- size of array
  VAR a[number + 1],   ---- array of integers
      b[number + 1],   ---- array of integers
      e[number + 1],   ---- array of integers
      clock,           ---- integer -variable to get time
      working,         ---- boolean -to stop execution
      counter,         ---- integer -number of operations done
      ch :

  SEQ
    -- initialize buffers and variables
    SEQ i = [ 1 FOR number ]
      SEQ
        a[i] := 3*i
        b[i] := 5*i
    SKIP
    counter := 0
    working := TRUE
    WHILE working
      ALT
        flag1 ? ch
          working := FALSE
        TIME ? clock
          SEQ
            SEQ i = [1 FOR number]
              e[i] := a[i] * b[i]
            counter := counter + number ---updates nr. of operations
    counterchan ! counter:
```

```
-- PROC inout.transfer1 (VALUE repetition,cpumode)
PROC inout.transfer1 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures iotransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC iotransfer1 (VALUE repetition,cpumode,CHAN done,opnumber)
  PROC iotransfer1 (VALUE repetition,cpumode, CHAN done, opnumber)=
  -- description
  --- ************************************************************
  --- It executes sequentially several parallel transfers using the
  --- input/output primitives to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes, repetition
  --- ************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR buffer0 [BYTE maxblock.size + 1]:
    VAR buffer1 [BYTE maxblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxblock.size]
        SEQ
          buffer0-BYTE i- := i\8
          buffer1-BYTE i- := i\8
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          block.size := sizetable[i]
          -- input and output handling
          -- input from two channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                in ? buffer0[BYTE k]
          SKIP
          -- input/output to two links
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                PAR
                  in ? buffer0[BYTE k]
                  out ! buffer1[BYTE k]
          SKIP
          -- input from three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                in ? buffer0[BYTE k]
          SKIP
          -- input/output to three links
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
```

```
            SEQ k = [1 FOR block.size]
              PAR
                in ? buffer0[BYTE k]
                out ! buffer1[BYTE k]
        SKIP
        -- input from four channels
        SEQ j = [1 FOR repetition]
          SEQ
            out ! 'a'
            SEQ k = [1 FOR block.size]
              in ? buffer0[BYTE k]
        SKIP
        -- output to four channels
        SEQ j = [1 FOR repetition]
          SEQ
            out ! 'a'
            SEQ k = [1 FOR block.size]
              out ! buffer0[BYTE k]
        SKIP
        -- all output and input in parallel
        SEQ j = [1 FOR repetition]
          SEQ
            out ! 'a'
            SEQ k = [1 FOR block.size]
              PAR
                in ? buffer0[BYTE k]
                out ! buffer1[BYTE k]
        SKIP
    SKIP
    IF
      -- cpumode NOT = '0' then get the number of computations done.
      cpumode <> '0'
        SEQ
          done ! 'a'
          opnumber ? number
      TRUE
        SKIP  :

-- main PROC inout.transfer1
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      iotransfer1 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      iotransfer1 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      iotransfer1 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      iotransfer1 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    iotransfer1 (repetition, cpumode, flag, counter):
```

```
-- PROC byte.slice.transfer1 (VALUE repetition,cpumode)
PROC byte.slice.transfer1 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures transfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC transfer1(VALUE repetition, cpumode, CHAN done, opnumber)
  PROC transfer1 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ****************************************************************
  --- It executes sequentially several parallel transfers using the
  --- BYTE.SLICE procedures to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes,
  --- repetition
  --- ****************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR buffer0 [BYTE maxblock.size + 1]:
    VAR buffer1 [BYTE maxblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxblock.size]
        SEQ
          buffer0-BYTE i- := i\8
          buffer1-BYTE i- := i\8
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          block.size := sizetable[i]
          -- input and output handling
          -- input from two channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- input/output to two channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
          -- input from three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- input/output to three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
```

```
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
          -- input from four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- output to four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.OUTPUT(out,buffer0,1,block.size)
          SKIP
          -- all output and input in parallel
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
      SKIP
      IF
        -- cpumode not = '0' then get the number of computations done.
        cpumode <> '0'
          SEQ
            done ! 'a'
            opnumber ? number
        TRUE
          SKIP  :

-- main byte.slice.transfer1
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      transfer1 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      transfer1 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      transfer1 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      transfer1 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    transfer1 (repetition, cpumode, flag, counter):
```

```
-- PROC int.transfer1 (VALUE repetition,cpumode)
PROC int.transfer1 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures intransfer,
--- plus, when applicable according to cpumode, one of the following:
---      cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,      ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC intransfer1 (VALUE repetition, cpumode, CHAN done, ...)
  PROC intransfer1 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ************************************************************
  --- It executes sequentially several parallel transfers of integers
  --- using the input/output primitives to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes,
  --- repetition
  --- ************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR wbuffer0 [maxwordblock.size + 1]:
    VAR wbuffer1 [maxwordblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxwordblock.size]
        SEQ
          wbuffer0[i] := i
          wbuffer1[i] := i
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        IF
          sizetable[i] < 4
            SKIP ---minimum number of bytes is 4 for integer transfer
          TRUE
            SEQ
              block.size := sizetable[i]
              -- input and output handling
              -- input from two channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    in ? wbuffer0[k]
              SKIP
              -- input/output to two links
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    PAR
                      in ? wbuffer0[k]
                      out ! wbuffer1[k]
              SKIP
              -- input from three channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    in ? wbuffer0[k]
```

```
                SKIP
                -- input/output to three links
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    SEQ k = [1 FOR (block.size/4)]
                      PAR
                        in ? wbuffer0[k]
                        out ! wbuffer1[k]
                SKIP
                -- input from four channels
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    SEQ k = [1 FOR (block.size/4)]
                      in ? wbuffer0[k]
                SKIP
                -- output to four channels
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    SEQ k = [1 FOR (block.size/4)]
                      out ! wbuffer0[k]
                SKIP
                -- all output and input in parallel
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    SEQ k = [1 FOR (block.size/4)]
                      PAR
                        in ? wbuffer0[k]
                        out ! wbuffer1[k]
                SKIP
    SKIP
    IF
      -- cpumode not = '0' then get the number of computations done.
      cpumode <> '0'
        SEQ
          done ! 'a'
          opnumber ? number
      TRUE
        SKIP  :

-- main  PROC int.transfer1
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      intransfer1 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      intransfer1 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      intransfer1 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      intransfer1 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    intransfer1 (repetition, cpumode, flag, counter):
```

```
-- PROC word.slice.transfer1 (VALUE repetition,cpumode)
PROC word.slice.transfer1 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures wordtransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC wordtransfer1 (VALUE repetition, cpumode, CHAN done, ...)
  PROC wordtransfer1 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ****************************************************************
  --- It executes sequentially several parallel transfers of integers
  --- using the WORD SLICE procedure to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes,
  --- repetition
  --- ****************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR wbuffer0 [ maxwordblock.size + 1]:
    VAR wbuffer1 [ maxwordblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxwordblock.size]
        SEQ
          wbuffer0[i] := i
          wbuffer1[i] := i
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        IF
          sizetable[i] < 4
            SKIP--- minimum number of bytes is 4 for integer transfer
          TRUE
            SEQ
              block.size := sizetable[i]
              -- input and output handling
              -- input from two channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
              SKIP
              -- input/output to two links
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  PAR
                    WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                    WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
              SKIP
              -- input from three channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
              SKIP
              -- input/output to three links
              SEQ j = [1 FOR repetition]
```

```
                  SEQ
                    out ! 'a'
                    PAR
                      WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                      WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
                SKIP
                -- input from four channels
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                SKIP
                -- output to four channels
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    WORD.SLICE.OUTPUT(out,wbuffer0,1,(block.size/4))
                SKIP
                -- all output and input in parallel
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    PAR
                      WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                      WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
                SKIP
        SKIP
        IF
          -- cpumode not = '0' then get the number of computations done.
          cpumode <> '0'
            SEQ
              done ! 'a'
              opnumber ? number
          TRUE
            SKIP  :

    -- main PROC word.slice.transfer1
    IF
      ((cpumode = '1') OR (cpumode = '2'))
        PAR
          wordtransfer1 (repetition, cpumode, flag, counter)
          cpubusysum (flag, counter)
      ((cpumode = '3') OR (cpumode = '4'))
        PRI PAR
          wordtransfer1 (repetition, cpumode, flag, counter)
          cpubusysum (flag, counter)
      ((cpumode = '5') OR (cpumode = '6'))
        PAR
          wordtransfer1 (repetition, cpumode, flag, counter)
          cpubusyprod (flag, counter)
      ((cpumode = '7') OR (cpumode = '8'))
        PRI PAR
          wordtransfer1 (repetition, cpumode, flag, counter)
          cpubusyprod (flag, counter)
      TRUE
        wordtransfer1 (repetition, cpumode, flag, counter):
-- procedure body transfer1.b003
SEQ
  answer := 'Y'
  WHILE  answer = 'Y'
    SEQ
      in ? ch
      in ? cpumode
      in ? repetition
      IF
        ch = 'A'
```

```
            inout.transfer1 (repetition,cpumode)
          ch = 'B'
            byte.slice.transfer1 (repetition,cpumode)
          ch = 'I'
            int.transfer1 (repetition,cpumode)
          ch = 'W'
            word.slice.transfer1 (repetition,cpumode)
          TRUE
            SKIP
        in ? answer :

--- ***************************************************************
--- END OF CODE IN TRANSPUTER 1 B003
--- ***************************************************************
```

```
-- TRANSPUTER2_B003.TDS
-- SC PROC transfer2.b003
-- PROC transfer2.b003 (CHAN in,out)
PROC transfer2.b003 (CHAN in,out) =
-- description
--- ****************************************************************
--- This is the outer procedure placed on transputer 2 . It contains
--- global variables and constants, and all procedures that run in this
--- transputer. It receives a construct type (ch), cpu mode (cpumode),
--- and number of times each communication sequence (repetition), and
--- calls accordingly one of the following procedures:
---         - io.transfer2,
---         - byte.slice.transfer2,
---         - int.transfer2 or
---         - word.slice.transfer2
--- ****************************************************************

  -- Link Definitions
  DEF link0in  = 4 :
  DEF link0out = 0 :
  DEF link1in  = 5 :
  DEF link1out = 1 :
  DEF link2in  = 6 :
  DEF link2out = 2 :
  DEF link3in  = 7 :
  DEF link3out = 3 :
  -- constant declarations
  DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                          1024, 1280, 2048, 4096, 8192, 10000 ]:
  DEF nr.of.sizes = 16:        --- as counted from above table
  DEF maxblock.size = 10000:   --- last from the above table
  DEF maxwordblock.size = maxblock.size/4:
  -- variable declarations
  VAR ch        : --- choice of the user in type of construct
  VAR answer    : --- choice of the user in continue
  VAR cpumode   : --- choice of the user in cpu operation concurrently
  VAR repetition: --- choice of the user in number of times to run
  -- SC PROC cpubusysum (CHAN flag1, counterchan)
  -- PROC cpubusysum (CHAN flag1,counterchan)
  PROC cpubusysum (CHAN flag1,counterchan)=
  -- description
  ---  ****************************************************************
  ---  It keeps the cpu working in parallel (time sharing) with link
  ---  transfers by doing sum operations . It Stops when it receives
  ---  a flag by the channel flag1 from the transfer procedure that is
  ---  being executed concurrently. It Outputs by channel counterchan
  ---  the number of operations done.
  ---  ****************************************************************

    VAR a,b,e,
        working,
        counter,
        ch :
    SEQ
      counter := 0
      working := TRUE
      TIME ? a
      WHILE working
        ALT
          flag1 ? ch
            working := FALSE
          TIME ? b
            SEQ
              e := a + b
              counter := counter + 1
      counterchan ! counter:
```

```
-- descriptor
-- code
-- SC PROC cpubusyprod (CHAN flag1,counterchan)
-- PROC cpubusyprod (CHAN flag1,counterchan)
PROC cpubusyprod (CHAN flag1,counterchan)=
-- description
---  ****************************************************************
---  It keeps the cpu working in parallel(time sharing) with the link
---  transfers by doing array multiplications. It stops when receives
---  a flag by the channel flag1 from the transfer procedure, that is
---  being executed concurrently. It outputs by channel counterchan
---  the number of operations done.
---  ****************************************************************
  -- constants and variable declarations
  DEF number = 100:   ---- size of array
  VAR a[number + 1],  ---- array of integers
      b[number + 1],  ---- array of integers
      e[number + 1],  ---- array of integers
      clock,          ---- integer -variable to get time
      working,        ---- boolean -to stop execution
      counter,        ---- integer -number of operations done
      ch :

  SEQ
    -- initialize buffers and variables
    SEQ i = [ 1 FOR number ]
      SEQ
        a[i] := 3*i
        b[i] := 5*i
    SKIP
    counter := 0
    working := TRUE
    WHILE working
      ALT
        flag1 ? ch
          working := FALSE
        TIME ? clock
          SEQ
            SEQ i = [1 FOR number]
              e[i] := a[i] * b[i]
            counter := counter + number ---updates nr. of operations
    counterchan ! counter:
```

```
-- PROC inout.transfer2 (VALUE repetition,cpumode)
PROC inout.transfer2 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures iotransfer,
--- plus, when applicable according to cpumode, one of the following:
---      cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,      ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC iotransfer2 (VALUE repetition, cpumode, CHAN done, counter)
  PROC iotransfer2 (VALUE repetition,cpumode,CHAN done, opnumber)=
    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR buffer0 [BYTE maxblock.size + 1]:
    VAR buffer1 [BYTE maxblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxblock.size]
        SEQ
          buffer0-BYTE i- := i\8
          buffer1-BYTE i- := i\8
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          block.size := sizetable[i]
          -- input and output handling
          -- input from three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                in ? buffer0[BYTE k]
          SKIP
          -- output to three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                PAR
                  in ? buffer0[BYTE k]
                  out ! buffer1[BYTE k]
          SKIP
          -- input from four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                in ? buffer0[BYTE k]
          SKIP
          -- output to four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                out ! buffer0[BYTE k]
          SKIP
          -- all output and input in parallel
          SEQ j = [1 FOR repetition]
            SEQ
```

```
                out ! 'a'
                SEQ k = [1 FOR block.size]
                   PAR
                      in ? buffer0[BYTE k]
                      out ! buffer1[BYTE k]
          SKIP
      SKIP
      IF
        -- cpumode NOT = '0' then get the number of computations done.
        cpumode <> '0'
          SEQ
            done ! 'a'
            opnumber ? number
        TRUE
          SKIP  :

-- main inout.transfer2
IF
   ((cpumode = '1') OR (cpumode = '2'))
     PAR
       iotransfer2 (repetition, cpumode, flag, counter)
       cpubusysum (flag, counter)
   ((cpumode = '3') OR (cpumode = '4'))
     PRI PAR
       iotransfer2 (repetition, cpumode, flag, counter)
       cpubusysum (flag, counter)
   ((cpumode = '5') OR (cpumode = '6'))
     PAR
       iotransfer2 (repetition, cpumode, flag, counter)
       cpubusyprod (flag, counter)
   ((cpumode = '7') OR (cpumode = '8'))
     PRI PAR
       iotransfer2 (repetition, cpumode, flag, counter)
       cpubusyprod (flag, counter)
   TRUE
     iotransfer2 (repetition, cpumode, flag, counter):
```

```
-- PROC byte.slice.transfer2 (VALUE repetition,cpumode)
PROC byte.slice.transfer2 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures iotransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC transfer2 (VALUE repetition, cpumode, CHAN done, counter)
  PROC transfer2 (VALUE repetition,cpumode,CHAN done, opnumber)=
    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR buffer0 [BYTE maxblock.size + 1]:
    VAR buffer1 [BYTE maxblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxblock.size]
        SEQ
          buffer0-BYTE i- := i\8
          buffer1-BYTE i- := i\8
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          block.size := sizetable[i]
          -- input from three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- input/output to three channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
          -- input from four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- output to four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.OUTPUT(out,buffer0,1,block.size)
          SKIP
          -- all output and input in parallel
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
```

```
      SKIP
      IF
        -- cpumode NOT = '0' then get the number of computations done.
        cpumode <> '0'
          SEQ
            done ! 'a'
            opnumber ? number
        TRUE
          SKIP  :

-- main byte.slice.transfer2
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      transfer2 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      transfer2 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      transfer2 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      transfer2 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    transfer2 (repetition, cpumode, flag, counter):
```

```
-- PROC int.transfer2 (VALUE repetition,cpumode)
PROC int.transfer2 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures intransfer,
--- plus, when applicable according to cpumode, one of the following:
---      cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC intransfer2 (VALUE repetition, cpumode, CHAN done,...)
  PROC intransfer2 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description.io
  --- ****************************************************************
  --- It executes sequentially several parallel transfers of integers
  --- using the input/output primitives to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes,
  --- repetition
  --- ****************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR wbuffer0 [maxwordblock.size + 1]:
    VAR wbuffer1 [maxwordblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxwordblock.size]
        SEQ
          wbuffer0[i] := i
          wbuffer1[i] := i
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        IF
          sizetable[i] < 4
            SKIP ---minimum number of bytes is 4 for integer transfer
          TRUE
            SEQ
              block.size := sizetable[i]
              -- input and output handling
              -- input from three channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    in ? wbuffer0[k]
              SKIP
              -- input/output to three links
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    PAR
                      in ? wbuffer0[k]
                      out ! wbuffer1[k]
              SKIP
              -- input from four channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    in ? wbuffer0[k]
```

```
                SKIP
                -- output to four channels
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    SEQ k = [1 FOR (block.size/4)]
                      out ! wbuffer0[k]
                SKIP
                -- all output and input in parallel
                SEQ j = [1 FOR repetition]
                  SEQ
                    out ! 'a'
                    SEQ k = [1 FOR (block.size/4)]
                      PAR
                        in ? wbuffer0[k]
                        out ! wbuffer1[k]
                SKIP
    SKIP
    IF
      -- cpumode NOT = '0' then get the number of computations done.
      cpumode <> '0'
        SEQ
          done ! 'a'
          opnumber ? number
      TRUE
        SKIP  :

-- main int.transfer2
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      intransfer2 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      intransfer2 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      intransfer2 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      intransfer2 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    intransfer2 (repetition, cpumode, flag, counter):
```

```
-- PROC word.slice.transfer2 (VALUE repetition,cpumode)
PROC word.slice.transfer2 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures wordtransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC wordtransfer2 (VALUE repetition, cpumode, CHAN done, ...)
  PROC wordtransfer2 (VALUE repetition,cpumode,CHAN done, opnumber)=
    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR wbuffer0 [ maxwordblock.size + 1]:
    VAR wbuffer1 [ maxwordblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxwordblock.size]
        SEQ
          wbuffer0[i] := i
          wbuffer1[i] := i
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        IF
          sizetable[i] < 4
            SKIP ---minimum number of bytes is 4 for integer transfer
          TRUE
            SEQ
              block.size := sizetable[i]
              -- input and output handling
              -- input from three channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
              SKIP
              -- input/output to three links
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  PAR
                    WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                    WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
              SKIP
              -- input from four channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
              SKIP
              -- output to four channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.OUTPUT(out,wbuffer0,1,(block.size/4))
              SKIP
              -- all output and input in parallel
              SEQ j = [1 FOR repetition]
                SEQ
```

```
                        out ! 'a'
                        PAR
                          WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                          WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
                    SKIP
          SKIP
          IF
            -- cpumode NOT = '0' then get the number of computations done.
            cpumode <> '0'
              SEQ
                done ! 'a'
                opnumber ? number
            TRUE
              SKIP  :

      -- main word.slice.transfer2
      IF
        ((cpumode = '1') OR (cpumode = '2'))
          PAR
            wordtransfer2 (repetition, cpumode, flag, counter)
            cpubusysum (flag, counter)
        ((cpumode = '3') OR (cpumode = '4'))
          PRI PAR
            wordtransfer2 (repetition, cpumode, flag, counter)
            cpubusysum (flag, counter)
        ((cpumode = '5') OR (cpumode = '6'))
          PAR
            wordtransfer2 (repetition, cpumode, flag, counter)
            cpubusyprod (flag, counter)
        ((cpumode = '7') OR (cpumode = '8'))
          PRI PAR
            wordtransfer2 (repetition, cpumode, flag, counter)
            cpubusyprod (flag, counter)
        TRUE
          wordtransfer2 (repetition, cpumode, flag, counter):
  -- procedure body transfer2.b003
  SEQ
    answer := 'Y'
    WHILE  answer = 'Y'
      SEQ
        in ? ch
        in ? cpumode
        in ? repetition
        IF
          ch = 'A'
            inout.transfer2 (repetition,cpumode)
          ch = 'B'
            byte.slice.transfer2 (repetition,cpumode)
          ch = 'I'
            int.transfer2 (repetition,cpumode)
          ch = 'W'
            word.slice.transfer2 (repetition,cpumode)
          TRUE
            SKIP
        in ? answer   :

--- ************************************************************************
--- END OF CODE IN TRANSPUTER 2
--- ************************************************************************
```

```
-- TRANSPUTER3_B003.TDS
-- SC PROC transfer3.b003
-- PROC transfer3.b003 (CHAN in,out)
PROC transfer3.b003 (CHAN in,out) =
-- description
--- ********************************************************************
--- This is the outer procedure placed on transputer 3 . It contains
--- global variables and constants, and all procedures that run in this
--- transputer. It receives a construct type (ch), cpu mode (cpumode),
--- and number of times each communication sequence (repetition), and
--- calls accordingly one of the following procedures:
---         - io.transfer3,
---         - byte.slice.transfer3,
---         - int.transfer3 or
---         - word.slice.transfer3
--- ********************************************************************

  -- Link Definitions
  DEF link0in  = 4 :
  DEF link0out = 0 :
  DEF link1in  = 5 :
  DEF link1out = 1 :
  DEF link2in  = 6 :
  DEF link2out = 2 :
  DEF link3in  = 7 :
  DEF link3out = 3 :
  -- constant declarations
  DEF sizetable = TABLE [ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512,
                          1024, 1280,  2048, 4096, 8192, 10000]:
  DEF nr.of.sizes = 16:          --- as counted from above table
  DEF maxblock.size = 10000:     --- last from the above table
  DEF maxwordblock.size = maxblock.size/4:
  -- variable declarations
  VAR ch         :   --- choice of the user in type of construct
  VAR answer     :   --- choice of the user in continue
  VAR cpumode    :  --- choice of the user in cpu operation concurrently
  VAR repetition:   --- choice of the user in number of times to run
  -- SC PROC cpubusysum (CHAN flag1, counterchan)
  -- PROC cpubusysum (CHAN flag1,counterchan)
  PROC cpubusysum (CHAN flag1,counterchan)=
  -- description.cpu
  ---  ****************************************************************
  ---  It keeps the cpu working in parallel (time sharing) with link
  ---  transfers by doing sum operations . It Stops when it receives
  ---  a flag by the channel flag1 from the transfer procedure that is
  ---  being executed concurrently. It Outputs by channel counterchan
  ---  the number of operations done.
  ---  ****************************************************************
  --- ********************************************************************

    VAR a,b,e,
        working,
        counter,
        ch :

    SEQ
      counter := 0
      working := TRUE
      TIME ? a
      WHILE working
        ALT
          flag1 ? ch
            working := FALSE
          TIME ? b
            SEQ
              e := a + b
              counter := counter + 1
      counterchan ! counter:
```

```
-- descriptor
-- code
-- SC PROC cpubusyprod (CHAN flag1,counterchan)
-- PROC cpubusyprod (CHAN flag1,counterchan)
PROC cpubusyprod (CHAN flag1,counterchan)=
-- description
---  ****************************************************************
---  It keeps the cpu working in parallel(time sharing) with the link
---  transfers by doing array multiplications. It stops when receives
---  a flag by the channel flag1 from the transfer procedure, that is
---  being executed concurrently. It outputs by channel counterchan
---  the number of operations done.
---  ****************************************************************
  -- constants and variable declarations
  DEF number = 100:   ---- size of array
  VAR a[number + 1],  ---- array of integers
      b[number + 1],  ---- array of integers
      e[number + 1],  ---- array of integers
      clock,          ---- integer -variable to get time
      working,        ---- boolean -to stop execution
      counter,        ---- integer -number of operations done
      ch :

  SEQ
    -- initialize buffers and variables
    SEQ i = [ 1 FOR number ]
      SEQ
        a[i] := 3*i
        b[i] := 5*i
    SKIP
    counter := 0
    working := TRUE
    WHILE working
      ALT
        flag1 ? ch
          working := FALSE
        TIME ? clock
          SEQ
            SEQ i = [1 FOR number]
              e[i] := a[i] * b[i]
            counter := counter + number ---updates nr. of operations
    counterchan ! counter:
```

```
-- PROC inout.transfer3 (VALUE repetition,cpumode)
PROC inout.transfer3 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures iotransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************


  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC iotransfer3 (VALUE repetition, cpumode, CHAN done,...)
  PROC iotransfer3 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ****************************************************************
  --- It executes sequentially several parallel transfers of bytes
  --- using the input/output primitives to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes,
  --- repetition
  --- ****************************************************************


    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR buffer0 [BYTE maxblock.size + 1]:
    VAR buffer1 [BYTE maxblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxblock.size]
        SEQ
          buffer0-BYTE i- := i\8
          buffer1-BYTE i- := i\8
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          block.size := sizetable[i]
          -- input from four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                in ? buffer0[BYTE k]
          SKIP
          -- output to four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                out ! buffer0[BYTE k]
          SKIP
          -- all output and input in parallel
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              SEQ k = [1 FOR block.size]
                PAR
                  in ? buffer0[BYTE k]
                  out ! buffer1[BYTE k]
          SKIP
      SKIP
      IF
        -- cpumode not='0' then get the number of computations done.
        cpumode <> '0'
```

```
          SEQ
            done ! 'a'
            opnumber ? number
        TRUE
          SKIP  :

-- main  inout.transfer3
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      iotransfer3 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      iotransfer3 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      iotransfer3 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      iotransfer3 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    iotransfer3 (repetition, cpumode, flag, counter):
```

```
-- PROC byte.slice.transfer3 (VALUE repetition,cpumode)
PROC byte.slice.transfer3 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures transfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************


  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC transfer3 (VALUE repetition, cpumode, CHAN done, ...)
  PROC transfer3 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ****************************************************************
  --- It executes sequentially several parallel transfers of BYTES
  --- using the BYTE.SLICE procedures to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes,
  --- repetition
  --- ****************************************************************


    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR buffer0 [BYTE maxblock.size + 1]:
    VAR buffer1 [BYTE maxblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxblock.size]
        SEQ
          buffer0-BYTE i- := i\8
          buffer1-BYTE i- := i\8
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        SEQ
          block.size := sizetable[i]
          -- input from four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.INPUT(in,buffer0,1,block.size)
          SKIP
          -- output to four channels
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              BYTE.SLICE.OUTPUT(out,buffer0,1,block.size)
          SKIP
          -- all output and input in parallel
          SEQ j = [1 FOR repetition]
            SEQ
              out ! 'a'
              PAR
                BYTE.SLICE.INPUT(in,buffer0,1,block.size)
                BYTE.SLICE.OUTPUT(out,buffer1,1,block.size)
          SKIP
      SKIP
      IF
        -- cpumode not='0' then get the number of computations done.
        cpumode <> '0'
          SEQ
            done ! 'a'
            opnumber ? number
```

```
        TRUE
          SKIP  :

-- main  byte.slice.transfer3
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      transfer3 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      transfer3 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      transfer3 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      transfer3 (repetition, cpumode, flag, counter)
      cpubusyprod (flag, counter)
  TRUE
    transfer3 (repetition, cpumode, flag, counter):
```

```
-- PROC int.transfer3 (VALUE repetition,cpumode)
PROC int.transfer3 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures intransfer,
--- plus, when applicable according to cpumode, one of the following:
---      cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,      ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC intransfer3 (VALUE repetition, cpumode, CHAN done, ...)
  PROC intransfer3 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ****************************************************************
  --- It executes sequentially several parallel transfers of integers
  --- using the input/output primitives to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes,
  --- repetition
  --- ****************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR wbuffer0 [maxwordblock.size + 1]:
    VAR wbuffer1 [maxwordblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxwordblock.size]
        SEQ
          wbuffer0[BYTE i] := i
          wbuffer1[BYTE i] := i
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        IF
          sizetable[i] < 4
            SKIP ---minimum number of bytes is 4 for integer transfer
          TRUE
            SEQ
              block.size := sizetable[i]
              -- input from four channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    in ? wbuffer0[k]
              SKIP
              -- output to four channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    out ! wbuffer0[k]
              SKIP
              -- all output and input in parallel
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  SEQ k = [1 FOR (block.size/4)]
                    PAR
                      in ? wbuffer0[k]
                      out ! wbuffer1[k]
              SKIP
```

```
      SKIP
      IF
        -- cpumode not='0' then get the number of computations done.
        cpumode <> '0'
          SEQ
            done ! 'a'
            opnumber ? number
        TRUE
          SKIP  :

-- main int.transfer3
IF
  ((cpumode = '1') OR (cpumode = '2'))
    PAR
      intransfer3 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '3') OR (cpumode = '4'))
    PRI PAR
      intransfer3 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '5') OR (cpumode = '6'))
    PAR
      intransfer3 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  ((cpumode = '7') OR (cpumode = '8'))
    PRI PAR
      intransfer3 (repetition, cpumode, flag, counter)
      cpubusysum (flag, counter)
  TRUE
    intransfer3 (repetition, cpumode, flag, counter) :
```

```
-- PROC word.slice.transfer3 (VALUE repetition,cpumode)
PROC word.slice.transfer3 (VALUE repetition,cpumode)=
-- description
--- ****************************************************************
--- Initializes the buffers and executes the procedures wordtransfer,
--- plus, when applicable according to cpumode, one of the following:
---     cpubusy.prod or cpubusy.sum.
--  Uses global constant maxblock.size
--- ****************************************************************

  CHAN flag,     ---  flags the cpu to stop
       counter : ---  return the number of operations cpu did
  -- PROC wordtransfer3 (VALUE repetition, cpumode, CHAN done,...)
  PROC wordtransfer3 (VALUE repetition,cpumode,CHAN done, opnumber)=
  -- description
  --- ****************************************************************
  --- It executes sequentially several parallel transfers of integers
  --- using the WORD.SLICE procedures to/from transputer root.
  --- It uses the global constants:  sizetable, nr.of.sizes,
  --- repetition
  --- ****************************************************************

    -- variable declarations
    VAR block.size,
        number,  --- the number of operations done by the cpu.
        ch[4]:
    VAR wbuffer0 [ maxwordblock.size + 1]:
    VAR wbuffer1 [ maxwordblock.size + 1]:

    SEQ
      -- initialize buffers
      SEQ i = [1 FOR maxwordblock.size]
        SEQ
          wbuffer0[BYTE i] := i
          wbuffer1[BYTE i] := i
      SKIP
      SEQ i = [0 FOR nr.of.sizes]
        IF
          sizetable[i] < 4
            SKIP---minimum number of bytes is 4 for integer transfer
          TRUE
            SEQ
              block.size := sizetable[i]
              -- input and output handling
              -- input from four channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
              SKIP
              -- output to four channels
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  WORD.SLICE.OUTPUT(out,wbuffer0,1,(block.size/4))
              SKIP
              -- all output and input in parallel
              SEQ j = [1 FOR repetition]
                SEQ
                  out ! 'a'
                  PAR
                    WORD.SLICE.INPUT(in,wbuffer0,1,(block.size/4))
                    WORD.SLICE.OUTPUT(out,wbuffer1,1,(block.size/4))
              SKIP
      SKIP
      IF
```

```
          -- cpumode not='0' then get the number of computations done.
          cpumode <> '0'
            SEQ
              done ! 'a'
              opnumber ? number
          TRUE
            SKIP  :

    -- main  word.slice.transfer3
    IF
      ((cpumode = '1') OR (cpumode = '2'))
        PAR
          wordtransfer3 (repetition, cpumode, flag, counter)
          cpubusysum (flag, counter)
      ((cpumode = '3') OR (cpumode = '4'))
        PRI PAR
          wordtransfer3 (repetition, cpumode, flag, counter)
          cpubusysum (flag, counter)
      ((cpumode = '5') OR (cpumode = '6'))
        PAR
          wordtransfer3 (repetition, cpumode, flag, counter)
          cpubusyprod (flag, counter)
      ((cpumode = '7') OR (cpumode = '8'))
        PRI PAR
          wordtransfer3 (repetition, cpumode, flag, counter)
          cpubusyprod (flag, counter)
      TRUE
        wordtransfer3 (repetition, cpumode, flag, counter):
-- main  transfer3.b003
SEQ
  answer := 'Y'
  WHILE  answer = 'Y'
    SEQ
      in ? ch
      in ? cpumode
      in ? repetition
      IF
        ch = 'A'
          inout.transfer3 (repetition,cpumode)
        ch = 'B'
          byte.slice.transfer3 (repetition,cpumode)
        ch = 'I'
          int.transfer3 (repetition,cpumode)
        ch = 'W'
          word.slice.transfer3 (repetition,cpumode)
        TRUE
          SKIP
      in ? answer   :
```

```
--- ********************************************************************
-- configuration
--- ********************************************************************

-- Link Definitions
DEF link0in  = 4 :
DEF link0out = 0 :
DEF link1in  = 5 :
DEF link1out = 1 :
DEF link2in  = 6 :
DEF link2out = 2 :
DEF link3in  = 7 :
DEF link3out = 3 :

DEF root = 100:
CHAN pipein[4],pipeout[4]:

PLACED PAR
  -- PROCESSOR ROOT
  PROCESSOR root
    PLACE pipein[0]    AT link0in  :
    PLACE pipeout[0]   AT link0out :
    PLACE pipein[1]    AT link1in  :
    PLACE pipeout[1]   AT link1out :
    PLACE pipein[2]    AT link2in  :
    PLACE pipeout[2]   AT link2out :
    PLACE pipein[3]    AT link3in  :
    PLACE pipeout[3]   AT link3out :
    hostproc (pipein[0],pipein[1],pipein[2],pipein[3],
              pipeout[0],pipeout[1],pipeout[2],pipeout[3])

  -- PROCESSOR 0
  PROCESSOR 0
    PLACE pipein[0]    AT link0out :
    PLACE pipeout[0]   AT link0in  :
    transfer0.b003 (pipeout[0],pipein[0])

  -- PROCESSOR 1
  PROCESSOR 1
    PLACE pipein[1]    AT link0out :
    PLACE pipeout[1]   AT link0in  :
    transfer1.b003 (pipeout[1],pipein[1])

  -- PROCESSOR 2
  PROCESSOR 2
    PLACE pipein[2]    AT link0out :
    PLACE pipeout[2]   AT link0in  :
    transfer2.b003 (pipeout[2],pipein[2])

  -- PROCESSOR 3
  PROCESSOR 3
    PLACE pipein[3]    AT link0out :
    PLACE pipeout[3]   AT link0in  :
    transfer3.b003 (pipeout[3],pipein[3])
```

APPENDIX F
PROGRAM TEST LINEARITY

```
-- header.occ
--    ************************************************************
--    *   Title :   Test Performance Linearity
--    *   Version : 2
--    *   Mod :  0
--    *   Author :  Jose Vanni Filho, Lcdr., Brazilian  Navy
--    *   Date :  June, 5th,1987
--    *   Programming Language : OCCAM 1
--    *   Compiler : IMS D 600 - TDS
--    *   Brief Description :  This program mapped in 17
--    *     transputers shows us the capability of the
--    *     transputer in linear increase of performance
--    *     with the increase of the number of processors.
--    *
--    ************************************************************
-- Brief Description
---  This program runs in 17 transputers:
---  transputer Root - prompts the user and triggers the other
---                     transputers .
---                  - times the whole process execution
---                  - receives the results and send to the screen
---  transputers 00,10,20,30 - execute two processes in parallel:
---                  - routes the trigger and the results,
---                  - executes the procedure counter
---  transputers 01,02,03,11,12,13,21,22,23,31,32,33 (12)
---                  - executes the procedure counter only

-- PROGRAM testlinearity17
-- testlinearity
-- SC PROC hostproc
-- PROC hostproc
PROC hostproc (CHAN A,B,C,D,E,F,G,H) =
  -- global definitions (partial)
  -- Constants Definitions
  DEF port    = 0: --- assign the i/o port of the B001 to the terminal
  DEF baud    = 11:--- set the baud.rate to 9600 bps
  DEF null    = 0: --- constantly used ASCII values
  DEF bell    = 7:
  DEF tab     = 9:
  DEF lf      = 10:
  DEF cr      = 13:
  DEF esc     = 27:
  DEF sp      = 32:

  -- Channels Definitions
  CHAN Parameters AT  0 :
  CHAN Screen :  --- AT 1: |This placements cannot be done in TDS. The
  CHAN Keyboard: --- AT 2: |terminal.driver already takes care of that
  -- Link Definitions
  DEF link0out = 0 :
  DEF link1out = 1 :
  DEF link2out = 2 :
  DEF link3out = 3 :
  DEF link0in  = 4 :
  DEF link1in  = 5 :
  DEF link2in  = 6 :
  DEF link3in  = 7 :
  -- File Handler Control Values
```

```
DEF ClosedOK           =  -1 :
DEF CloseFile          =  -2 :
DEF EndBuffer          =  -3 :
DEF EndFile            =  -4 :
DEF EndName            =  -5 :
DEF EndParameterString =  -6 :
DEF EndRecord          =  -7 :
DEF NextRecord         =  -9 :
DEF OpenedOK           = -10 :
DEF OpenForRead        = -11 :
DEF OpenForWrite       = -12 :
-- library.occ   (partial)
-- io_routines.occ  (partial)
-- Summary of i/o PROCs
--- PROC new.line       generates a CR and a LF
--- PROC write.string   outputs a string to the screen, byte by byte
--- PROC clear.screen   clears the screen and home the cursor
--- PROC write.number   displays an integer value in the screen
-- PROC new.line
--- ************************************************************
--- Jumps to a new line on the screen
--- ************************************************************
PROC new.line =
  SEQ
    Screen ! cr;lf;EndBuffer :
-- PROC write.string (VALUE string[])
--- ************************************************************
--- Writes a given string to the screen, in a byte by byte fashion
--- ************************************************************
PROC write.string (VALUE string[]) =
--- ************************************************************
  SEQ
    SEQ i = [1 FOR string[BYTE 0]]
      Screen ! string[BYTE i]
    Screen ! EndBuffer :
--PROC clear.screen
--- ************************************************************
--- Clears the screen.
--- ************************************************************
PROC clear.screen =
  SEQ
    Screen ! esc; '-'; '2'; 'J'; EndBuffer --- clear screen sequence
    Screen ! esc; '-'; 'H' :               --- home cursor
-- PROC write.number (VALUE number)
--- ************************************************************
--- This PROC outputs a signed integer value to the screen   *
--- ************************************************************
PROC write.number(VALUE number) =
  VAR output[16], count, x:
  SEQ
    x:= number
    count:= 0
    IF
      --  handle special cases
      x=0
        Screen ! '0'
      x<0
        SEQ
          Screen ! '-'
          x:=-x
      TRUE
        SKIP
    WHILE x>0
      --  construct number
      SEQ
        output[count] := (x   10) + '0'
```

```
          count := count + 1
          x:= x/10
      WHILE count > 0
        --  output number
        SEQ
          count := count-1
          Screen ! output[count]
      SKIP:
--- ****************************************************************
-- utilities.occ
--- ****************************************************************

-- PROC tick.to.time (VALUE start, stop, board.type)
--- ****************************************************************
--- Receives start and stop time and board type and outputs
--- the elapsed time in hours, minutes, seconds and milliseconds
--- ****************************************************************
PROC tick.to.time (VALUE start, stop, board.type) =
  --- board.type = 0 ---->  VAX VMS
  --- board.type = 1 ---->  B001
  --- board.type = 2 ---->  B002
  --- board.type = 31---->  B003 ( high priority )
  --- board.type = 32---->  B003 ( low priority )
  --- board.type = 4 ---->  B004
  ---
  -- constant definitions
  DEF vax.sec    =10000000 :   --- hundreds of nsec/second
  DEF vax.mili   =   10000 :   --- hundreds of nsec/millisecond
  DEF b001.sec   =  625000 :   --- # of 1.6 microsec/second
  DEF b001.mili  =     625 :   --- # of 1.6 microsec/millisecond
  DEF b003h.sec  = 1000000 :   --- # of microsec/second
  DEF b003h.mili =    1000 :   --- # of microsec/millisecond
  DEF b003l.sec  =   15625 :   --- # of 64 microsec/second
  DEF b003l.mili =      16 :   --- # of 64 microsec/millisecond
  DEF max.number.of.ticks = 2147483648 : --- maximum integer (2**31)

  VAR elapsed.tick :
  VAR factor1, factor2 :
  VAR msec, tot.sec, sec, min, hr :
  SEQ
    IF
      board.type = 0                          --- VAX VMS
        SEQ
          factor1 := vax.sec
          factor2 := vax.mili
      board.type  = 1                         --- B001
        SEQ
          factor1 := b001.sec
          factor2 := b001.mili
      board.type  = 2                         --- B002
        SKIP                  --- will be implemented in the future
      board.type  = 31                        --- B003 in high priority
        SEQ
          factor1 := b003h.sec
          factor2 := b003h.mili
      board.type  = 32                        --- B003 in low priority
        SEQ
          factor1 := b003l.sec
          factor2 := b003l.mili
      board.type  = 4                         --- B004
        SKIP                --- will be implemented in the future
    elapsed.tick := stop - start
    IF
      elapsed.tick < 0
```

```
        elapsed.tick := elapsed.tick + max.number.of.ticks
      TRUE
        SKIP
    tot.sec := elapsed.tick/factor1
    hr      := tot.sec/3600
    min     := (tot.sec\3600)/60
    sec     := tot.sec\60
    msec    := (elapsed.tick factor1)/factor2
    -- output time to screen
    write.number (hr)
    write.string (" hr ")
    write.number (min)
    write.string (" min ")
    write.number(sec)
    write.string (" sec ")
    write.number(msec)
    write.string (" msec") :
-- PROC capitalize (VAR ch)
--- ************************************************************
--- capitalizes any lower case character into upper case
--- ************************************************************
PROC capitalize (VAR ch) =
  DEF delta =('a' - 'A') :
                                --- A ---> 65
                                --- a ---> 97      ASCII values
                                --- z ---> 122
  SEQ
    IF
      (ch <= 'z') AND (ch >= 'a')
        ch := ch - delta
      TRUE
        SKIP :
-- link placements
CHAN hostin0 AT link0in:
CHAN hostin1 AT link1in:
CHAN hostin2 AT link2in:
CHAN hostin3 AT link3in:
CHAN hostout0 AT link0out:
CHAN hostout1 AT link1out:
CHAN hostout2 AT link2out:
CHAN hostout3 AT link3out:
-- PROC terminal driver
************************************************************
The terminal driver is the one provided by the manufacturer
for the b001 board and therefore is not included.
************************************************************
```

```
-- PROC user.interface
---************************************************************
--- Receive flag from the user and triggers the network
--- Receive results from the network and output to the screen
---************************************************************
PROC user.interface =
  -- local constant and variable declaration
  DEF tot = 16 : --- number of transputers
  VAR ch:        --- flag from the user
  VAR result[tot]:
  VAR startimeroot, endtimeroot:   --- timers for the root
  VAR startime[tot], endtime[tot]: --- timers for the 16 transputers

  SEQ
    write.string(" Type any character to start ")
    Keyboard ? ch
    Screen ! ch
    new.line
    TIME ? startimeroot
    PAR
      -- send flags
      hostout0 ! ch
      hostout1 ! ch
      hostout2 ! ch
      hostout3 ! ch
      -- receive results
      SEQ i = [0 FOR 4]
        PAR
          hostin0 ? result[i];startime[i];endtime[i]
          hostin1 ? result[i+4];startime[i+4];endtime[i+4]
          hostin2 ? result[i+8];startime[i+8];endtime[i+8]
          hostin3 ? result[i+12];startime[i+12];endtime[i+12]
      SKIP
    TIME ? endtimeroot
    -- send results to the screen
    SEQ j = [0 FOR tot]
      SEQ
        write.string ("Transputer ")
        write.number (j)
        Screen ! sp; sp
        write.number (result[j])
        Screen ! sp; sp
        tick.to.time (startime[j],endtime[j],32)
        new.line
    SKIP
    -- send total execution time to the screen
    new.line
    write.string (" Time to execute in parallel ")
    write.string (" with 17 transputers => ")
    tick.to.time (startimeroot,endtimeroot,1):

  PAR
    IMS.B001.terminal.driver(Keyboard,Screen,port,baud)
    user.interface :
--- ********************************************************************
--- End of code for transputer Root.
--- ********************************************************************
```

```
-- SC PROC Route
-- PROC Route (CHAN messagein, messageout, routeto1,...,VALUE k)
PROC route(CHAN messagein,messageout,routeto1,routeto2,routeto3,
                  echofrom1,echofrom2,echofrom3,VALUE k)=
  DEF i = 4 :     --- number of counter procedures
  VAR msg :       --- flag
  VAR results[i] :
  VAR startime[i],endtime[i]:  --- timers
  CHAN softin,softout:  --- soft channels declared for communication
                            --- with procedure counter.
  -- SC PROC counter
  -- PROC counter
  PROC counter (CHAN in,out, VALUE tnumber) =
  -- description
  ---*****************************************************************
  --- Sums up the first 100000 integers and add the transputer number
  --- to the total
  ---*****************************************************************
    DEF maxope = 100000:   --- number of operations done
    VAR ch,total :
    VAR startime3, endtime3:
    SEQ
      total := tnumber
      in ? ch
      TIME ? startime3
      SEQ i = [0 FOR maxope]
        total := total + i
      TIME ? endtime3
      out ! total;startime3;endtime3:
  -- descriptor
  -- code

  SEQ
    PAR
      counter (softout,softin,k)
      -- routing procedure
      SEQ
        messagein ? msg
        SEQ
          PAR
            routeto1 ! msg
            routeto2 ! msg
            routeto3 ! msg
          softout ! msg
          PAR
            echofrom1 ? results-0-;startime-0-;endtime-0-
            echofrom2 ? results-1-;startime-1-;endtime-1-
            echofrom3 ? results-2-;startime-2-;endtime-2-
            softin ? results-3-; startime-3-;endtime-3-
    -- sending to the root results and timing
    SEQ i = [0 FOR 4]
      messageout ! results[i];startime[i];endtime[i]:
--- ****************************************************************
--- End of code for transputers Routers (00,10,20,30)
--- ****************************************************************
```

```
-- SC PROC counter
-- PROC counter (CHAN in,out, VALUE tnumber)
PROC counter (CHAN in,out, VALUE tnumber) =
-- description
---*************************************************************
--- Sums up the first 100000 integers and add the transputer number
--- to the total. Sends the result through channel out.
---*************************************************************

  DEF maxope = 100000:
  VAR ch,total :
  VAR startime, endtime:
  SEQ
    total := tnumber
    in ? ch
    TIME ? startime
    SEQ i = [0 FOR maxope]
      total := total + i
    TIME ? endtime
    out ! total;startime;endtime:
--- *************************************************************
--- End of code for transputers Leaves  (01,02,03,11,12,13,21,22,... )
--- *************************************************************
```

```
--- ********************************************************************
-- configuration
--- ********************************************************************

-- link definitions
DEF link0in = 4 :
DEF link0out = 0 :
DEF link1in = 5 :
DEF link1out = 1 :
DEF link2in = 6 :
DEF link2out = 2 :
DEF link3in = 7 :
DEF link3out = 3 :

DEF root = 100:
DEF totlinks = 32:
CHAN pipe[totlinks]:

PLACED PAR
  PROCESSOR root
    -- link placements and process assignment
    PLACE pipe[0] AT link0in :
    PLACE pipe[1] AT link0out :
    PLACE pipe[2] AT link1in  :
    PLACE pipe[3] AT link1out :
    PLACE pipe[4] AT link2in  :
    PLACE pipe[5] AT link2out :
    PLACE pipe[6] AT link3in  :
    PLACE pipe[7] AT link3out :

    hostproc (pipe[0],pipe[2],pipe[4],pipe[6],
              pipe[1],pipe[3],pipe[5],pipe[7])

  PLACED PAR j = [0 FOR 4]
    PROCESSOR 10*j
      -- link placements and process assignment
      PLACE pipe[2*j] AT link0out :
      PLACE pipe[(2*j)+1] AT link0in :

      PLACE pipe[8+(6*j)] AT link2in :
      PLACE pipe[9+(6*j)] AT link2out :
      PLACE pipe[10+(6*j)] AT link1in :
      PLACE pipe[11+(6*j)] AT link1out :
      PLACE pipe[12+(6*j)] AT link3in :
      PLACE pipe[13+(6*j)] AT link3out :

      route (pipe[(2*j)+1],pipe[2*j],pipe[9+(6*j)],pipe[11+(6*j)],
       pipe-13+(6*j)-,pipe-8+(6*j)-,pipe-10+(6*j)-,pipe-12+(6*j)-,10*j)

  PLACED PAR i = [0 FOR 4]
    PROCESSOR (10*i)+1
      -- link placements and process assignment
      PLACE pipe[8+(6*i)] AT link3out :
      PLACE pipe[9+(6*i)] AT link3in :

      counter(pipe[9+(6*i)],pipe[8+(6*i)],((10*i)+1))

  PLACED PAR i = [0 FOR 4]
    PROCESSOR (10*i)+2
      -- link placements and process assignment
      PLACE pipe[10+(6*i)] AT link1out :
      PLACE pipe[11+(6*i)] AT link1in :

      counter(pipe[11+(6*i)],pipe[10+(6*i)],((10*i)+2))

  PLACED PAR i = [0 FOR 4]
    PROCESSOR (10*i)+3
      -- link placements and process assignment
      PLACE pipe[12+(6*i)] AT link2out :
      PLACE pipe[13+(6*i)] AT link2in :

      counter(pipe[13+(6*i)],pipe[12+(6*i)],((10*i)+3))
```

APPENDIX G
TRANSPUTER PRODUCTS*

a. **Transputers**

- IMS T414B-G15S - 32 bit transputer - 15mhz
- IMS T414B-G20S - 32 bit transputer - 20mhz
- IMS T800B-G20S - 32 bit floating point transputer - 20mhz
- IMS T212A-G17S - 16 bit transputer - 17mhz
- IMS T212A-G20S - 16 bit transputer - 20mhz
- IMS M212B-G15S - Winchester and Floppy disk controller

b. **Evaluation Boards**

- IMS B002-2 - T 414 with 2MBytes DRAM with 2 x RS232
- IMS B003-1 - Described in Chapter I
- IMS B003-2 - 4 x T 414 - 20mhz each with 256KB DRAM
- IMS B004-4 - Described in Chapter I
- IMS B005-1 - M212 with 64kbytes SRAM, 20MB WINI, 640K Floppy
- IMS B006-1 - T212 with 64kbytes SRAM, and 2 x RS 232
- IMS B006-2 - T212 with 64kbytes SRAM, and 8 x T212 (8k SRAM)
- IMS B007-1 - Graphics Evaluation Board with 1 T414, 512k DRAM

c. **Digital Signal Processing**

- IMS A100-G20S - 32 Stage cascadeable signal processor

* All trademarks on this page are registered trademarks from INMOS Group of Companies, Bristol, UK.

LIST OF REFERENCES

1. Garret, D. R., *A Software System Implementation Guide and System Prototyping Facility for the MCORTEX Executive on the Real Time Cluster*, M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1986.

2. Weitzman, C., *Distributed Micro/Mini-computer Systems*, Prentice-Hall, New Jersey, 1980.

3. Peterson, J. & Silberchatz, A., *Operating Systems Concepts*, Second Edition, Addison-Wesley Publishing Co., Reading, Massachusetts, 1986.

4. Evin, B. , *Implementation of a Serial Delay Insertion Type Loop Communication for a Real Time Multitransputer System*, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

5. Selcuk, Z., *Implementation of a Serial Communication Process for a Fault Tolerant, Real Time, Multitransputer Operating System* M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1984.

6. Cordeiro, M. M., *Design, Implementation and Evaluation of an Operating System for a Transputer Network*, M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.

7. INMOS Limited, *Transputer Reference Manual*, October 1986.

8. Miller, Neil *Exploring Multiple Transputer Arrays*, INMOS Technical note 24, January 1987.

9. Kodres, U. R.,"Processing Efficiency of a Class of Multi-computer Systems", *International Journal of Mini and Micro-computers*, Volume 5, No.2, pp 28-33, 1983.

10. Wilson, P., "Occam Architecture Eases System Design - Part 1", *Computer Design*, Volume 22, No. 13, pp 107-110, November 1983.

11. Wilson, P., "Occam Architecture Eases System Design - Part 2", *Computer Design*, Volume 22, No. 14, pp 109-114, December 1983.

12. Pountain, D., *A Tutorial Introduction to Occam Programming*, 1985.

13. INMOS Limited, *Occam Programming System*, 1985.

14. INMOS Limited, *IMS D600 Transputer Development System*, 1985.

15. INMOS Limited, *IMS D701 Transputer Development System*, 1985.

16. INMOS Limited, *IMS B001 Evaluation Board User Manual*, 1985.

17. INMOS Limited, *IMS B003 Evaluation Board User Manual*, 1985.

18. INMOS Limited, *IMS B004 Evaluation Board User Manual*, 1985.

19. Halsall, F., *Introduction to Data Communications and Computer Networks* Addison-Wesley, Workingham, United Kingdom, 1985.

20. Cellary, W. and Stroinski, M., "Analysis of Methods of Computer Network Performance Measurement", *Performance of Computer Communication Systems*, Werner Bax and Harry Rudin Editors, North-Holland, 1984.

21. INMOS Limited, *IMS T800 Architecture* INMOS Technical note 6, Bristol, United Kingdom, 1986.

22. Naval Postgraduate School, Computer Science Department, *VAX/VMS Introduction*, by Bruce R. Montague, January 1983, revised June 1986.

BIBLIOGRAPHY

INMOS Corporation, *Compiler Writers Guide*, Draft, 1986.

INMOS Corporation, *Transputer America*, 1986.

INMOS Limited, *Product Information - The Transputer Family*, June 1986.

MacClennan, B. J., *Principles of Programming Languages: Design, Evaluation and Implementation*, CBS College Publishing, New York, 1983.

Stallings, W, *Computer Organization and Architecture*, Macmillan Publishing Company, New York, 1987

Mattos, P., *The Transputer Based Navigation System - An Example of Testing Embedded Systems*, INMOS Technical note 2, November 1986.

Mattos, P., *Program Design for Concurrent Systems* INMOS Technical note 5, December 1986.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
4.	Dr. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, CA 93943	3
5.	Dr. Daniel L. Davis, Code 52Dv Department of Computer Science Naval Postgraduate School Monterey, CA 93943	1
6.	Daniel Green, Code 20F Naval Surface Weapons Center Dahlgren, VA 22449	1
7.	Jerry Gaston, Code N24 Naval Surface Weapons Center Dahlgren, VA 22449	1
8.	CAPT. J. Hood, USN PMS 400B5 Naval Sea Systems Command Washington D.C. 20362	1
9.	RCA AEGIS Repository RCA Corporation Government Systems Division Mail Stop 127-327 Moorestown, NJ 08057	1
10.	Library (Code E33-05) Naval Surface Weapons Center Dahlgren, VA 22449	1

11. Dr. M. J. Gralia — 1
Applied Physics Laboratory
John Hopkins Road
Laurel, MD 20702

12. Dana Small, Code 8242 — 1
Naval Ocean Systems Center
San Diego, CA 92152

13. Estado Maior da Armada — 1
Brazilian Naval Commission
4706 Wisconsin Ave., N.W.
Washington, DC 20016

14. Diretoria de Ensino da Marinha — 1
Brazilian Naval Commission
4706 Wisconsin Ave., N.W.
Washington, DC 20016

15. Diretoria de Armamento e Comunicacoes da Marinha — 1
Brazilian Naval Commission
4706 Wisconsin Ave., N.W.
Washington, DC 20016

16. Instituto de Pesquisas da Marinha — 1
Brazilian Naval Commission
4706 Wisconsin Ave., N.W.
Washington, DC 20016

17. Instituto Militar de Engenharia — 1
Praia Vermelha, Urca
Rio de Janeiro, RJ
CEP 20000 , BRAZIL

18. Instituto Tecnologico da Aeronautica — 1
Sao Jose dos Campos, SP
CEP 11000 , BRAZIL

19. Pontificia Universidade Catolica — 1
R. Marques de Sao Vicente 225, Gavea
Rio de Janeiro, RJ
CEP 20000 , BRAZIL

20. Pete Wilson — 1
INMOS CORPORATION
P.O. Box 16000
Colorado Springs, CO 80935-16000

21. David May — 1
INMOS LTD.
1000 Aztec
West Almondsbury, Bristol, BS12 4SQ, UK

22. MAJ, USAF R. A. Adams, Code 52Ad — 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

23. LCDR. J. Vanni Filho, Br. Navy — 2
Brazilian Naval Commission (DACM)
4706 Wisconsin Ave., N.W.
Washington, DC 20016

24. LCDR. Gilberto F. Mota, Br. Navy — 1
Brazilian Naval Commission (DACM)
4706 Wisconsin Ave., N.W.
Washington, DC 20016

25. LT. M. M. Cordeiro, Br. Navy — 1
Brazilian Naval Commission (DACM)
4706 Wisconsin Ave., N.W.
Washington, DC 20016

18354/2

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CALIFORNIA 93943-5002

Thesis
F4275 Filho
c.1 Test and evaluation of the Transputer in a multi-transputer system.

thesF4275
Test and evaluation of the transputer in
3 2768 000 73608 6
DUDLEY KNOX LIBRARY